Automata-Theoretic Protocol Programming

Parallel Computation,
Threads and Their Interaction,
Optimized Compilation,
[at a] High Level of Abstraction

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus Prof. Mr. C.J.J.M. Stolker,
volgens besluit van het College voor Promoties
te verdedigen op donderdag 3 maart 2016
klokke 15:00 uur

door

Sung-Shik Theodorus Quirinus Jongmans
geboren te Seoul, Zuid Korea,
in 1987

Promotor
prof. dr. F. Arbab

Promotiecommissie

prof. dr. C. Baier	Technische Universität Dresden
prof. dr. F.S. de Boer	
dr. M.M. Bonsangue	
prof. dr. H.J. van den Herik (secretaris)	
prof. dr. J.N. Kok (voorzitter)	
dr. A. Silva	University College London

The work in this thesis has been carried out at CWI (Centrum Wiskunde & Informatica) and Universiteit Leiden, under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Contents

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-

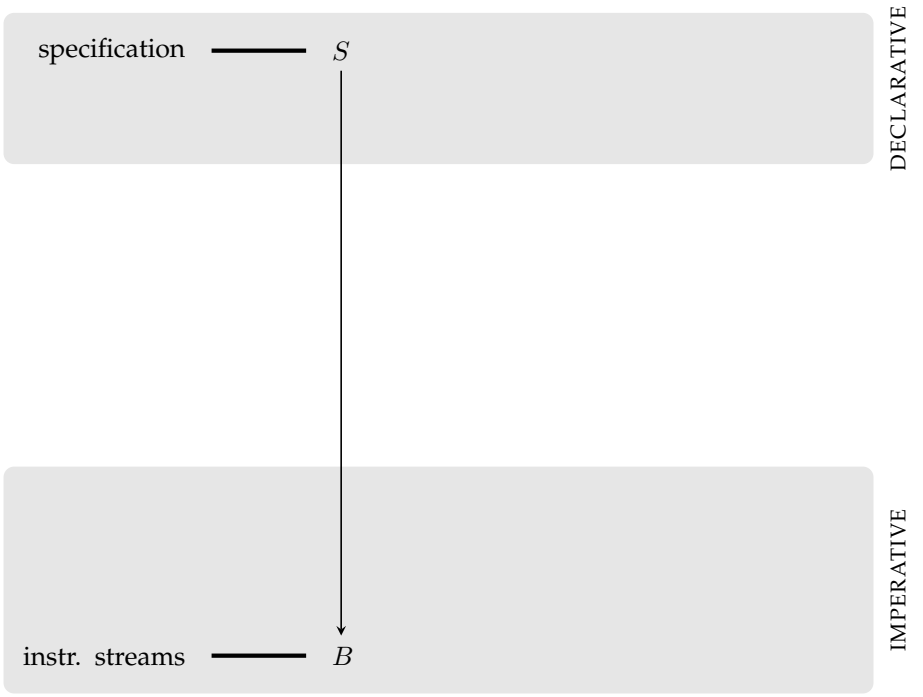


Figure 1.1: From a declarative specification S to its imperative implementation as instruction streams B.

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:

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

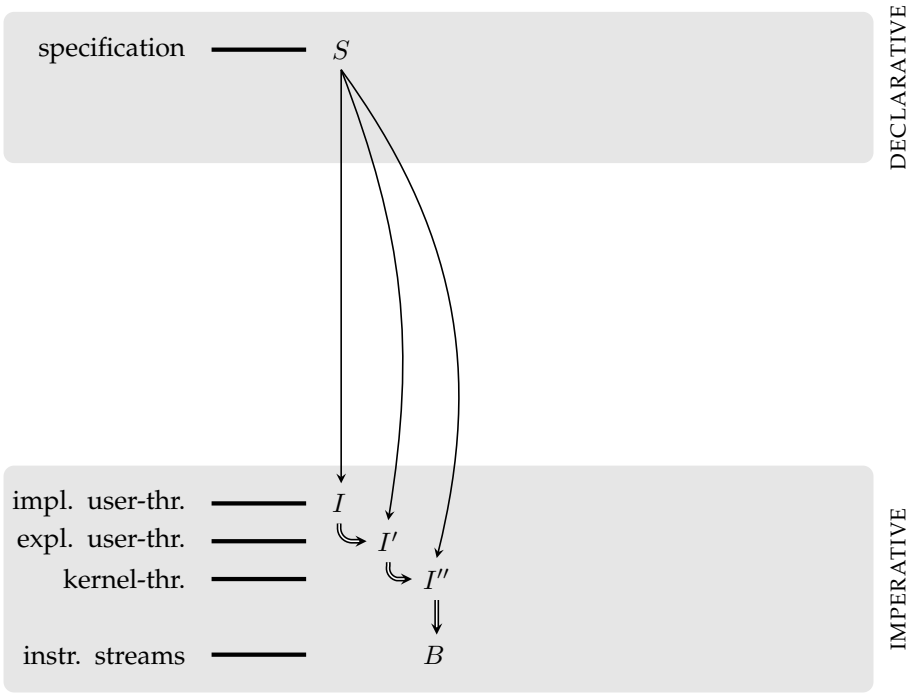


Figure 1.2: From a declarative specification S to its imperative implementation as instruction streams B, possibly past several intermediate imperative implementations I, I', and I'' at increasingly low levels of abstraction: implicit user-threads, explicit user-threads, and kernel-threads. Single arrows 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-

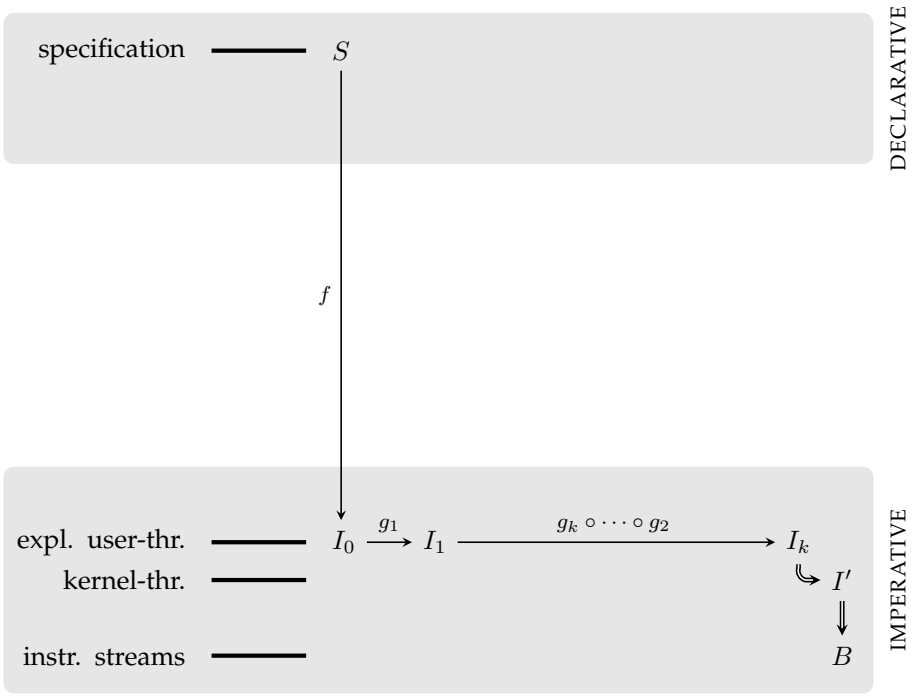


Figure 1.5: From a declarative specification S to its imperative implementation as instruction streams B, including k optimizations between intermediate imperative implementations $I_0, \ldots, I_k$ (at the explicit user-threads level). Single arrows represent manual labor by a software engineer; double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

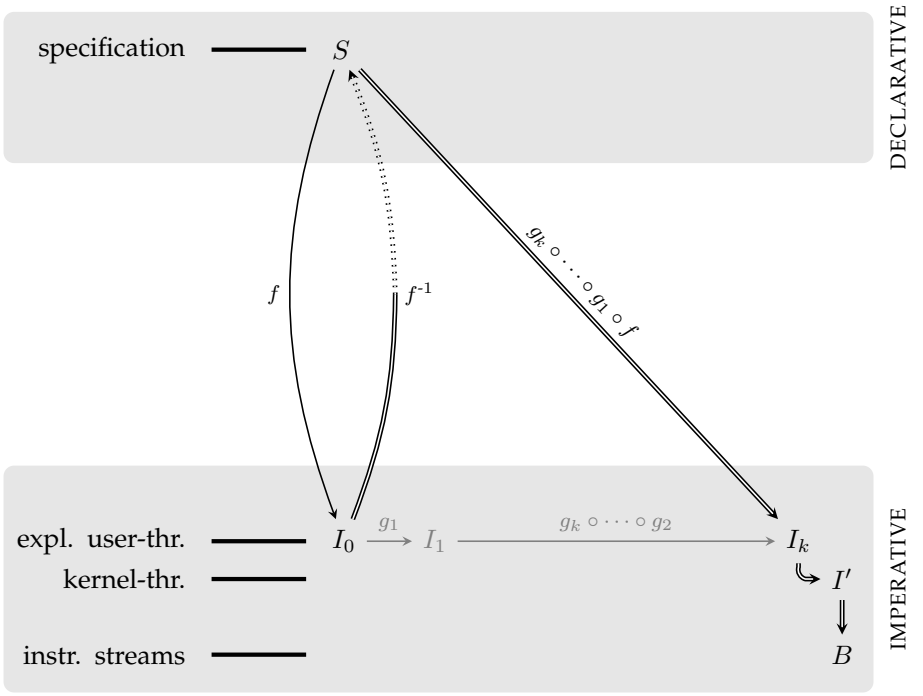


Figure 1.6: Irrecoverability of a specification S from its implementation I_0. Single arrows represent manual labor by a software engineer; double arrows represent automatic labor by a compiler.

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

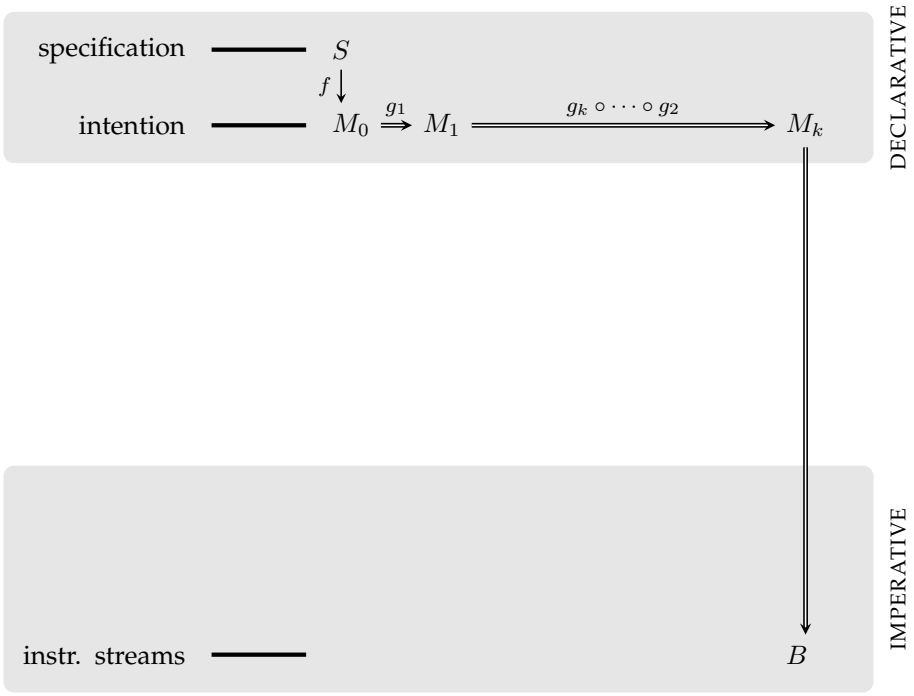


Figure 1.7: From a declarative specification S to its imperative implementation as instruction streams B, including k optimizations between intermediate declarative implementations $M_0, \ldots, M_k$ (at an intention-expressing level). Single arrows represent manual labor by a software engineer; double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

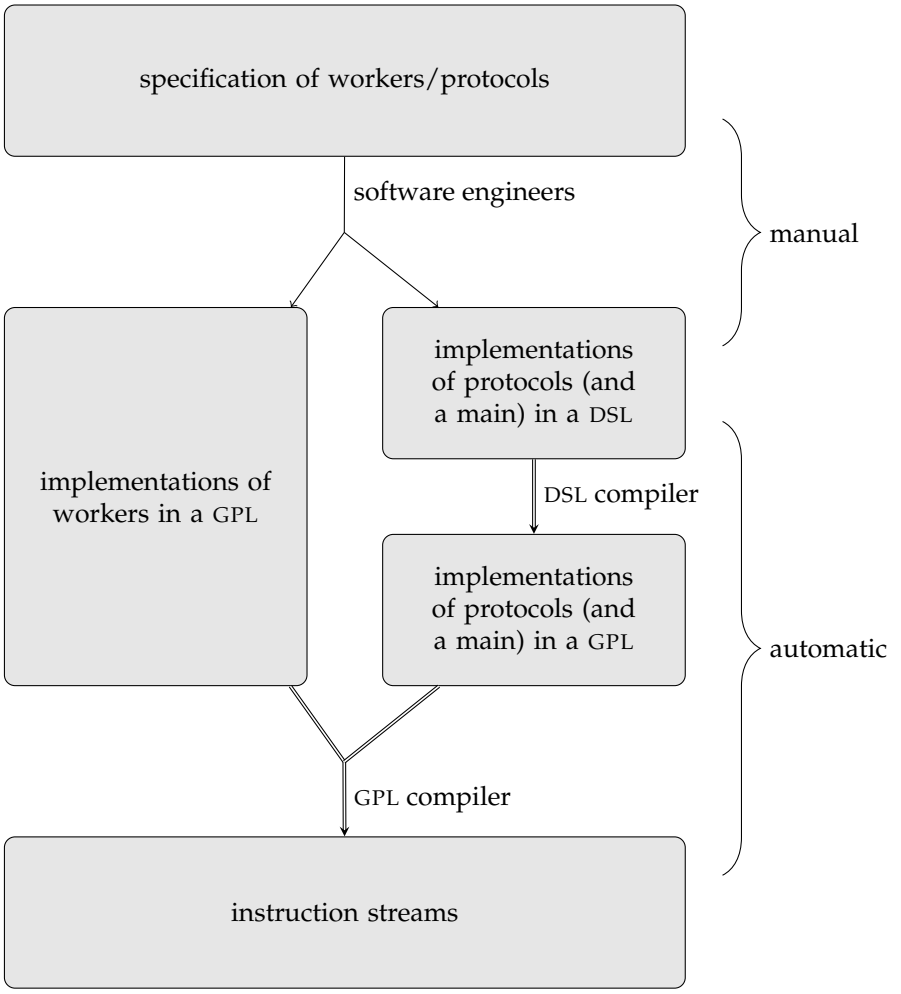


Figure 1.8: Software engineering with an interaction level of abstraction

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.

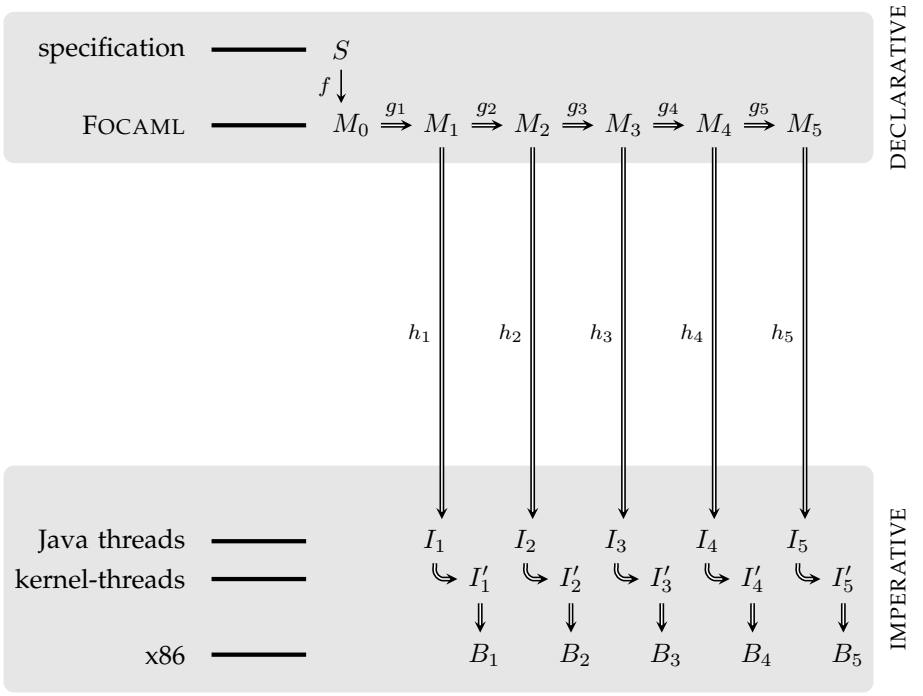


Figure 1.11: Instantiation of Figure 1.7

- 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.

Chapter 2

DSL for Interaction I: Semantics

As defined in Chapter 1, a true intention-expressing DSL for interaction provides constructs for implementing protocol specifications as first-class entities that preserve as much intention information from software engineers as possible. To achieve this, such constructs should very precisely capture the intention that people have when they use the word "protocol". In this thesis, I therefore commit myself to the Oxford Advanced American Dictionary, which informally defines "protocol" as follows:

> "**3** [countable] (*computing*) a set of rules that control the way data is sent between computers"—[Lea11, page 1179]

In this chapter, I present the semantics of a true intention-expressing DSL for interaction, whose programs satisfy this dictionary definition, replacing "computers" with "workers" according to terminology established in Chapter 1. In this DSL, significantly, *protocols* (i.e., admissible interaction, i.e., constraints on interaction as explained in Chapter 1) constitute the set of first-class entities (i.e., mathematical relations, concisely represented through automata, as explained in this chapter) and primary units of composition, built out of atomic protocols—interactions—which explicitly constrain the timing, ordering, and data-flows between actions. This strongly contrasts, for instance, process calculi, where *processes* constitute the set of first-class entities and primary units of composition, built of out atomic processes—actions—which only implicitly may (or may not!) induce interaction.

In Section 2.1, I present both elementary machinery for modeling "[...] the way data is sent between workers" and complementary machinery for modeling "a set of rules that control [...]". In Section 2.2, I briefly discuss a practical incarnation of the theoretical work in Section 2.1.

2.1 Theory

(I have not yet submitted the material in this section for publication.)

Interaction Languages

In this thesis, as explained in Chapter 1, interaction among workers occurs through blocking I/O operations with value-passing semantics on their ports. Workers may have access to multiple ports, each of which they may use for a different purpose or even in different protocols. Consequently, modeling protocols at the finer level of "interaction among ports" instead of at the coarser level of "interaction among workers" improves the accuracy of the resulting models. Indeed, in contrast to the latter approach, the former approach enables me to state precisely through which of a worker's ports certain interaction occurs. The compiler that I present in Chapter 4 actually requires this level of precision. Therefore, in what follows, I develop machinery for modeling "the way data is exchanged through ports" instead of "sent between workers".

As a first step, I formally define two essential ingredients: ports and data.

Definition 1 (ports)**.** *A port is an unstructured object.* $\mathbb{P}$ *denotes the set of all ports, ranged over by* p*.* $2^{\mathbb{P}}$ *denotes the set of all sets of ports, ranged over by* P, V*.* $2^{2^{\mathbb{P}}}$ *denotes the set of all sets of sets of ports, ranged over by* E*.* $2^{2^{2^{\mathbb{P}}}}$ *denotes the set of all sets of sets of sets of ports, ranged over by* G*.*

Definition 2 (data)**.** *A datum is an unstructured object.* $\mathbb{D}$ *denotes the set of all data, ranged over by* d*, such that* $\mathbb{P} \cap \mathbb{D} = \emptyset$*.*

Definition 3 (empty datum)**.** `nil` *is an unstructured object such that* $\texttt{nil} \notin \mathbb{D}$*.*

The extra condition in Definition 2 means that workers do not communicate their ports to other workers; the theory presented in this chapter does not support *mobility* as in π-calculus. In fact, in practice, the value-passing semantics of I/O operations inhibits this. Nevertheless, as explained in Chapter 1, nothing prevents a worker from sending a reference to a port *as a value* to another worker; and symmetrically, nothing prevents this other worker from interpreting and using this value as a reference to a port. Whenever software engineers set this up, however, they also take full responsibility for the consequences. Definition 3 asserts the existence of a distinguished empty datum. The exact content of $\mathbb{P}$ and $\mathbb{D}$ depends on the context of their use and formally does not matter much. For instance, if I use Java for writing worker subprograms, $\mathbb{D}$ contains all Java objects. Henceforth, I write elements of $\mathbb{P}$ in capitalized lower case sans-serif (e.g., A, B, C, In_1, Out_2), while I write elements of $\mathbb{D}$ in lower case monospace (e.g., `1`, `3.14`, `true`, `"foo"`). At this point, I do not yet distinguish input ports from output ports; this comes later, in the next subsection.

Out of ports and data, I construct *interaction letters*, so called for a reason that becomes clear shortly. Every interaction letter models one instance of interaction, in which particular data pass through particular ports. Formally, I

define an interaction letter λ as a partial function that associates a datum d with every port p in its domain, where $\lambda(p) = d$ means that d passes through p in the instance of interaction modeled by λ.

Definition 4 (interaction letters)**.** *An interaction letter is a partial function from ports to data.* $\mathbb{L}\text{ETT} = (\mathbb{P} \rightharpoonup \mathbb{D}) \setminus \emptyset$ *denotes the set of all interaction letters, ranged over by* λ*.* $2^{\mathbb{L}\text{ETT}}$ *denotes the set of all sets of interaction letters, ranged over by* Λ*.*

Suppose that one producer has access to port A, the other producer to port B, and the consumer to port C in the producers/consumer example in Chapter 1. In that case, interaction letter $\{\mathsf{A} \mapsto \texttt{"foo"}\}$ models an instance of interaction in which one producer exchanges datum `"foo"`—a string—through port A with its environment. Similarly, interaction letter $\{\mathsf{C} \mapsto \texttt{"foo"}\}$ models an instance of interaction in which the consumer exchanges `"foo"` through port C with its environment. Together, these two instances of interaction can model the asynchronous send/receive of `"foo"` from a producer (on port A) to the consumer (on port C). Similarly, interaction letter $\{\mathsf{A} \mapsto \texttt{"foo"}, \mathsf{C} \mapsto \texttt{"foo"}\}$ can model an instance of interaction in which a producer and the consumer synchronously exchange `"foo"`. The LateAsyncMerger2 protocol in Chapter 1 forbids such synchronous communication, though, so this third instance of interaction should never occur.

Out of interaction letters, I construct *interaction words*, so called for a reason that becomes clear shortly. Every interaction word models one chain of interaction, in which infinitely many instances of interaction follow each other. Formally, I define an interaction word w as an infinite sequence of nonempty interaction letters $\lambda_1\lambda_2\cdots$, where λ_i models the i-th instance of interaction in the chain of interaction modeled by w.

Definition 5 (interaction words)**.** *An interaction word is an infinite sequence of interaction letters.* $\mathbb{W}\text{ORD} = \mathbb{L}\text{ETT}^{\omega}$ *denotes the set of all interaction words, ranged over by* w*.*

Continuing the previous example, the following four interaction words model chains of interaction in which, from left to right, (i) only one producer communicates with the consumer, (ii) both producers communicate with the consumer (iii) both producers synchronously communicate with the consumer, and (iv) both producers nontransactionally communicate with the consumer:

(i)	(ii)	(iii)	(iv)
$\{\mathsf{A} \mapsto \texttt{"foo"}\}$	$\{\mathsf{A} \mapsto \texttt{"foo"}\}$	$\{\mathsf{A} \mapsto \texttt{"foo"}, \mathsf{C} \mapsto \texttt{"foo"}\}$	$\{\mathsf{A} \mapsto \texttt{"foo"}\}$
$\{\mathsf{C} \mapsto \texttt{"foo"}\}$	$\{\mathsf{C} \mapsto \texttt{"foo"}\}$	$\{\mathsf{B} \mapsto \texttt{"bar"}, \mathsf{C} \mapsto \texttt{"bar"}\}$	$\{\mathsf{B} \mapsto \texttt{"bar"}\}$
$\{\mathsf{A} \mapsto \texttt{"bar"}\}$	$\{\mathsf{B} \mapsto \texttt{"bar"}\}$	$\{\mathsf{A} \mapsto \texttt{"baz"}, \mathsf{C} \mapsto \texttt{"baz"}\}$	$\{\mathsf{C} \mapsto \texttt{"foo"}\}$
$\{\mathsf{C} \mapsto \texttt{"bar"}\}$	$\{\mathsf{C} \mapsto \texttt{"bar"}\}$	$\{\mathsf{B} \mapsto \texttt{"qux"}, \mathsf{C} \mapsto \texttt{"qux"}\}$	$\{\mathsf{C} \mapsto \texttt{"bar"}\}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

LateAsyncMerger2 forbids the chains of interaction modeled by (iii) and (iv).

Out of interaction words, I construct *interaction languages*, so called for a reason that becomes clear shortly. Every interaction language models a collection of chains of interaction. Formally, I define an interaction language $\mathcal{L}$ as a set of interaction words.

Definition 6 (interaction languages)**.** *An interaction language is a set of interaction words.* $\mathbb{L}\text{ANG} = 2^{\mathbb{W}\text{ORD}}$ *denotes the set of all interaction languages, ranged over by* $\mathcal{L}$*.*

Continuing the previous example, the following interaction language models all chains of interaction admitted by LateAsyncMerger2.

$$\left\{ w \;\middle|\; \begin{array}{l} w \in \mathbb{W}\text{ORD} \\ \textbf{and} \;\; \big[\big[i \bmod 2 = 0 \;\; \textbf{implies} \;\; \text{Dom}(w(i)) \subset \{\mathsf{A}, \mathsf{B}\}\big] \;\; \textbf{for all} \;\; i\big] \\ \textbf{and} \;\; \big[\big[i \bmod 2 = 1 \;\; \textbf{implies} \;\; \text{Dom}(w(i)) = \{\mathsf{C}\}\big] \;\; \textbf{for all} \;\; i\big] \end{array} \right\}$$

The machinery presented so far models the second component of the dictionary definition on page 27 as follows. First, an interaction word (i.e., a chain of interaction) models one infinite "way data is exchanged through ports" in one particular infinite run of a program. Then, an interaction language (i.e., a collection of chains of interaction) models a collection of infinite ways "data is exchanged through ports". One can model finite ways "data is exchanged through ports" by extending finite sequences to infinite sequences as usual.

Interaction letters, interaction words, and interaction languages go by different names in the literature. For instance, Izadi et al. call interaction letters *records*, interaction words *streams of records*, and interaction languages *languages of records* [IBC11, Iza11]. Alternatively, both Baier et al., Klein, and Klüppelholz et al. call interaction letters *concurrent* I/O *operations* and interaction words I/O *streams* [BBK+10, BBKK09a, BBKK09b, BKK11, KB09, KB10, Kle12, Klü12], while Arbab et al. call interaction words *scheduled data streams* [ABdBR07]. Each of those names refers to the same kind of mathematical object, though. Tuples of *timed data streams* (TDS) comprise a different but still related kind of mathematical object, originally introduced by Rutten and Arbab and later further developed by Arbab into *abstract behavior types* [AR03, Arb05]. Every tuple of TDSs contains one TDS for every port of interest. Every TDS, in turn, consists of two infinite sequences: a *time stream* of monotonically increasing real numbers and a *data stream* of data. A TDS for a port p subsequently models that the i-th datum in the data stream flows through p at the time represented by the i-th real number in the time stream. Consequently, tuples of TDSs contain not only information about the order in which instances of interaction take place but also more precise timing information. For instance, if I need to extract the full instance of interaction that occurs at time 3.14, I check for every port (i) which index 3.14 has in that port's time stream, and if such an index indeed exists, (ii) which datum occurs in that port's data stream at that index.

Constraint Automata

If an interaction language contains exactly those interaction words that model the chains of interaction admitted by some protocol, this interaction language indeed models that protocol. To model protocols in terms of interaction languages, thus, I need a method of concisely specifying the content of interaction languages. This brings me to the first component of the dictionary definition on page 27: I need machinery for modeling "a set of rules that control [the second component]". I intend to capture such "a set of rules" that a protocol consists of with an *automaton* of some kind, and in particular, with that automaton's *transition relation*. By constructing this automaton such that it *accepts* interaction words, each of its transitions effectively models one of the *state*ful "rules that control the way data is exchanged through ports".

Naively, I may adopt the set of all interaction letters as my alphabet and use Büchi automata as interaction language acceptors. In this approach, every infinite sequence of transitions straightforwardly corresponds to exactly one interaction word. However, such an *explicit representation* of interaction letters on transitions has a problem: if the set of all data $\mathbb{D}$ contains infinitely many elements, so does the set of all interaction letters $\mathbb{L}\text{ETT}$. In that case, the resulting Büchi automata have infinite transition relations, which I cannot account for.

Instead of labeling a transition $\boldsymbol{t}$ with an interaction letter, I label $\boldsymbol{t}$ with a *symbolic representation* of a possibly infinite set of interaction letters Λ. Every such a representation consists of two elements: a *synchronization constraint* and a *data constraint*. A synchronization constraint specifies the domain of every interaction letter in Λ. This models which ports participate in every instance of interaction modeled by Λ. Formally, I define a synchronization constraint as a set of ports. A data constraint specifies two things. First, it specifies to which data every interaction letter in Λ maps the ports in its domain. This models which data pass through which ports in every instance of interaction modeled by Λ. Second, it specifies the content of *memory cells* before and after firing $\boldsymbol{t}$. This models how internal buffers in a protocol evolve. Shortly, I define a data constraint as a formula in a first-order calculus with variables, constants, functions, and relations [Rau10a]. Before I can do so, however, I first need to introduce other machinery.

I start by formally defining memory cells.

> **Definition 7** (memory cells)**.** *A memory cell is an unstructured object. $\mathbb{M}$ denotes the set of all memory cells, ranged over by m. $2^{\mathbb{M}}$ denotes the set of all sets of memory cells, ranged over by M.*

The exact content of $\mathbb{M}$ depends on the context of its use and does not matter much. Henceforth, I write elements of $\mathbb{M}$ in lower case sans-serif (e.g., x, buff_1).

Out of memory cells and data, I construct *memory snapshots*. Every memory snapshot models the content of memory cells in some time instant. Formally, I define a memory snapshot μ as a partial function that associates a datum d to every memory cell m in its domain, where $\mu(m) = d$ means that m contains d in the time instant modeled by μ.

Definition 8 (memory snapshots)**.** *A memory snapshot is a partial function from memory cells to data.* $\mathbb{S}\text{NAPSH} = \mathbb{M} \rightharpoonup \mathbb{D}$ *denotes the set of all memory snapshots, ranged over by* μ*.*

Suppose that the producers and the consumer use buffer `x` for temporary storage of their data in the producers/consumer example in Chapter 1. In that case, $\{\mathtt{x} \mapsto \mathtt{0}\}$ models the initial content of the buffer (i.e., an arbitrarily selected datum), while $\{\mathtt{x} \mapsto \mathtt{"foo"}\}$ models the content of the buffer after a producer has sent `"foo"` to the consumer.

I proceed by defining variables in the calculus, called *data variables*. Every data variable models a container for data. For instance, ports can hold data, so every port serves as a data variable in the calculus. Similarly, memory cells can hold data, but the meaning of "to hold" differs in this case. Ports hold data only *during* an instance of interaction (i.e., transiently, in passing). In contrast, memory cells hold data also *before* and *after* an instance of interaction. Consequently, in the context of data variables, a memory cell before an instance of interaction and the same memory cell after that instance have a different identity. After all, the content of the memory cell may have changed in between. Therefore—inspired by notation from Petri nets [Rei85]—for every memory cell m, both ${}^\bullet m$ and $m^\bullet$ serve as data variables: ${}^\bullet m$ refers to the datum in m before an instance of interaction, while $m^\bullet$ refers to the datum in m after that instance. I abbreviate sets $\{{}^\bullet m \mid m \in M\}$ and $\{m^\bullet \mid m \in M\}$ as ${}^\bullet M$ and $M^\bullet$.

Definition 9 (data variables)**.** *A data variable is an object* x *generated by the following grammar:*

$$x ::= p \mid {}^\bullet m \mid m^\bullet \qquad \textit{(data variables)}$$

$\mathbb{X}$ *denotes the set of all data variables.* $2^{\mathbb{X}}$ *denotes the set of all sets of data variables, ranged over by* X*.*

I assign meaning to data variables with *data assignments*.

Definition 10 (data assignments)**.** *A data assignment is a partial function from data variables to data.* $\mathbb{A}\text{SSIGNM} = \mathbb{X} \rightharpoonup \mathbb{D}$ *denotes the set of all data assignments, ranged over by* σ*.* $2^{\mathbb{A}\text{SSIGNM}}$ *denotes the set of all sets of data assignments, ranged over by* Σ*.*

I proceed by defining constants, functions, and predicates in the calculus. To avoid excessive machinery—but at the cost of formal precision—I do not distinguish constant, function, and predicate symbols from their *interpretation* as data, functions on data, and relations on data [Rau10a]. Instead, I directly refer to data, *data functions*, and *data relations*.

Definition 11 (data functions)**.** *A data function is a function from tuples of data to data.* $\mathbb{F} = \bigcup\{\mathbb{D}^k \to \mathbb{D} \mid \mathbf{true}\}$ *denotes the set of all data functions, ranged over by* f*.*

Definition 12 (data relations)**.** *A data relation is a relation on tuples of data.* $\mathbb{R} = \bigcup\{2^{\mathbb{D}^k} \mid \mathbf{true}\}$ *denotes the set of all data relations, ranged over by* R*.*

Henceforth, I write elements of $\mathbb{F}$ in camel case monospace (e.g., `divByThree`, `inc`), while I write elements of $\mathbb{R}$ in capitalized camel case monospace (e.g., `Odd`, `SmallerThan`).

Out of data variables, data, and data functions, I construct *data terms*. Every data term represents a datum. This models an operation on (some of) the data involved in an instance of interaction.

Definition 13 (data terms)**.** *A data term is an object* t *generated by the following grammar:*

$$t ::= x \mid d \mid f(t_1, \ldots, t_{k\geq 1}) \qquad \textit{(data terms)}$$

$\mathbb{TERM}$ *denotes the set of all data terms.* $2^{\mathbb{TERM}}$ *denotes the set of all sets of data terms, ranged over by* T*.*

Henceforth, let $<_{\mathbb{TERM}}$ denote some strict total order on $\mathbb{TERM}$.

Given a data assignment whose domain includes at least the data variables in a data term t, one can *evaluate* t to a datum. (To evaluate t, additionally, every data function application in t must have the right number of inputs: the *arity* of a data function and its number of inputs must match. Henceforth, I tacitly assume that this always holds true.)

Definition 14 (evaluation)**.** $\mathsf{eval} : \mathbb{ASSIGNM} \times \mathbb{TERM} \to \mathbb{D} \cup \{\mathtt{nil}\}$ *denotes the function defined by the following equations:*

$$\mathsf{eval}_\sigma(x) = \begin{cases} \sigma(x) & \textbf{if } x \in \mathrm{Dom}(\sigma) \\ \mathtt{nil} & \textbf{otherwise} \end{cases}$$

$$\mathsf{eval}_\sigma(d) = d$$

$$\mathsf{eval}_\sigma(f(t_1, \ldots, t_k)) = \begin{cases} f(\mathsf{eval}_\sigma(t_1), \ldots, \mathsf{eval}_\sigma(t_k)) & \textbf{if } \begin{bmatrix} \mathsf{eval}_\sigma(t_1) \neq \mathtt{nil} \\ \textbf{and } \cdots \textbf{ and} \\ \mathsf{eval}_\sigma(t_k) \neq \mathtt{nil} \end{bmatrix} \\ \mathtt{nil} & \textbf{otherwise} \end{cases}$$

Out of data terms, data relations, and data variables, I finally construct a first-order calculus of data constraints. Although this calculus supports existential quantification, it does not support universal quantification for two reasons. First, universal quantification seems only marginally useful in this thesis (i.e., I do not miss the extra expressiveness that it would provide). More importantly, however, inclusion of universal quantification would complicate computing a particular normal form of data constraints in Chapter 6.

$$\frac{}{\sigma \models \top} \quad (2.1) \qquad \frac{\begin{bmatrix} m \in M \ \textbf{implies} \\ \sigma \models {}^{\bullet}m = m^{\bullet} \end{bmatrix} \ \textbf{for all} \ m}{\sigma \models \mathtt{K}(M)} \quad (2.2)$$

$$\frac{\mathsf{eval}_\sigma(t_1) = \mathsf{eval}_\sigma(t_2) \neq \mathtt{nil}}{\sigma \models t_1 = t_2} \quad (2.3) \qquad \frac{(\mathsf{eval}_\sigma(t_1), \ldots, \mathsf{eval}_\sigma(t_k)) \in R}{\sigma \models R(t_1, \ldots, t_k)} \quad (2.4)$$

$$\frac{\sigma \models \chi_1 \ \textbf{and} \ \sigma \models \chi_2}{\sigma \models \chi_1 \wedge \chi_2} \quad (2.5) \qquad \frac{\sigma \models \chi_1 \ \textbf{or} \ \sigma \models \chi_2}{\sigma \models \chi_1 \vee \chi_2} \quad (2.6)$$

$$\frac{\begin{array}{l} \mathsf{Free}(a) \subseteq \mathrm{Dom}(\sigma) \\ \textbf{and} \ \sigma \not\models a \end{array}}{\sigma \models \neg a} \quad (2.7) \qquad \frac{\sigma \models \phi[d/x] \ \textbf{for some} \ d}{\sigma \models \exists x.\phi} \quad (2.8)$$

$$\frac{\sigma \models \phi_1 \ \textbf{and} \ \cdots \ \textbf{and} \ \sigma \models \phi_k}{\sigma \models \phi_1 \wedge \cdots \wedge \phi_k} \quad (2.9) \qquad \frac{\sigma \models \phi_1 \ \textbf{or} \ \cdots \ \textbf{or} \ \sigma \models \phi_k}{\sigma \models \phi_1 \vee \cdots \vee \phi_k} \quad (2.10)$$

Figure 2.1: Addendum to Definition 16

Definition 15 (data constraints)**.** *A data constraint is an object ϕ generated by the following grammar:*

$$\begin{array}{lll} M & ::= & \textit{any subset of } \mathbb{M} \\ a & ::= & \bot \mid \top \mid \mathtt{K}(M) \mid t = t \mid R(t_1, \ldots, t_{k\geq 1}) \qquad \textit{(data atoms)} \\ \ell & ::= & a \mid \neg a \qquad \textit{(data literals)} \\ \chi & ::= & \ell \mid \chi \wedge \chi \mid \chi \vee \chi \qquad \textit{(data formulas)} \\ \phi & ::= & \chi \mid \exists x.\phi \mid \phi_1 \wedge \cdots \wedge \phi_{k\geq 2} \mid \phi_1 \vee \cdots \vee \phi_{k\geq 2} \qquad \textit{(data constraints)} \end{array}$$

$\mathbb{DC}$ denotes the set of all data constraints. $2^{\mathbb{DC}}$ denotes the set of all sets of data constraints, ranged over by Φ.

To simplify some of the proofs later in this thesis, the grammar features multiary conjunction and disjunction in addition to their binary versions. Also, negation cannot occur freely but only in data literals, because free occurrences of negation seriously complicate data constraint normalization.

Henceforth, let $<_{\mathbb{DC}}$ denote a strict total order on $\mathbb{DC}$, let $\bigwedge\Phi$ denote the *unique* multiary conjunction of the data constraints in Φ under $<_{\mathbb{DC}}$, and let $\bigvee\Phi$ similarly denote a unique multiary disjunction.

Every data constraint characterizes a set of interaction letters—its semantics—through an *entailment relation*. Let $\phi[t/x]$ denote data constraint ϕ with data term t substituted for every occurrence of data variable x (in a capture-free way).

Definition 16 (entailment)**.** $\models \subseteq \mathbb{ASSIGNM} \times \mathbb{DC}$ *denotes the smallest relation induced by the rules in Figure 2.1.*

Contradiction, tautology, (multiary) conjunction, and (multiary) disjunction have standard semantics [Rau10a]. Negation $\neg a$ means that, despite all free variables in a having a value, a does not hold true; the extra condition on the free variables in a ensures the *monotonicity* of entailment. Data atom $\mathtt{K}(M)$ means that every memory cell in M keeps the same value before and after an instance of interaction. Data atom $t_1 = t_2$ means that t_1 and t_2 evaluate to the same datum. Typical examples include $p_1 = p_2$ (i.e., the same datum passes through ports p_1 and p_2), $p = m^\bullet$ (i.e., the datum that passes through port p enters the buffer modeled by memory cell m), and $p = {}^\bullet m$ (i.e., the datum in the buffer modeled by memory cell m exits that buffer and passes through port p). Tautology $\top$ means that it does not matter which data flow through which ports.

Henceforth, let $\Rightarrow$ and $\equiv$ denote the implication relation and the equivalence relation on data constraints, derived from $\models$ in the usual way [Rau10a]. Furthermore, let $\mathsf{Variabl}(\phi)$ denote the set of data variables in ϕ, let $\mathsf{Free}(\phi)$ denote its set of *free* data variables, and let $\mathsf{Bound}(\phi)$ denote its multiset of *bound* data variables (i.e., $\mathsf{Bound}(\phi)$ contains as many occurrences of x as the number of quantifiers that bind x in ϕ).

Let X denote a set of data variables. I call a data constraint ϕ *good* under X if (i) ϕ has no free data variables outside X and (ii) the structure of ϕ either allows for moving all existential quantifiers outward, in a semantics-preserving way, or already satisfies such a *prenex normal form*. More precisely, to move existential quantifiers outward, (ii.a) ϕ must have no bound data variables in X and (ii.b) every existential quantifier in ϕ must bind a unique data variable. Typically, X contains all data variables for ports and memory cells in (a transition in) a constraint automaton $\boldsymbol{a}$. By subsequently requiring that all data constraints in $\boldsymbol{a}$ come from $\mathsf{Good}(X)$, condition (i) ensures that the protocol modeled by $\boldsymbol{a}$ cannot affect interaction on ports and memory cells outside its own scope. Condition (ii) plays a role in data constraint normalization, which I discuss in more detail in Chapter 6.

Definition 17 (goodness)**.** $\mathsf{Good} : 2^{\mathbb{X}} \to 2^{\mathbb{DC}}$ *denotes the function defined by the following equation:*

$$\mathsf{Good}(X) = \left\{ \phi \;\middle|\; \begin{array}{l} \mathsf{Free}(\phi) \subseteq X \;\; \textbf{and} \;\; \mathsf{Bound}(\phi) \cap X = \emptyset \\ \textbf{and} \;\; |\mathsf{Bound}(\phi)| = |\mathsf{Bound}(\phi) \cap \mathbb{X}| \end{array} \right\} \\ \cup \{\phi \mid \mathsf{Free}(\phi) \subseteq X \;\; \textbf{and} \;\; \phi = \exists x_1 \ldots \exists x_l . (\chi_1 \wedge \ldots \wedge \chi_k)\}$$

To understand the previous definition, recall that $\mathsf{Bound}(\phi)$ denotes a multiset, whereas $\mathbb{X}$ denotes an ordinary set. Then, observe that $\mathsf{Bound}(\phi) \cap \mathbb{X}$ contains only the distinct elements in $\mathsf{Bound}(\phi)$. Thus, if the multiset $\mathsf{Bound}(\phi)$ contains more elements than the ordinary set $\mathsf{Bound}(\phi) \cap \mathbb{X}$, at least two existential quantifiers in ϕ bind the same data variable; condition (ii.b) forbids this.

I proceed by defining *first-order constraint automata with memory*, each of which models a protocol, usually called just "constraint automata" in this thesis. Formally, I define a constraint automaton $\boldsymbol{a}$ as a tuple consisting of a set of states Q, a triple of three sets of ports $(P^{\text{all}}, P^{\text{in}}, P^{\text{out}})$, a set of memory cells M, a transition relation $\longrightarrow$, and an initial *configuration* (q^0, μ^0). Set P^{all} contains all ports that participate in the protocol modeled by $\boldsymbol{a}$, while P^{in} and P^{out} contain only its input ports and its output ports (where "input" and "output" qualify ports from the protocol perspective). Although P^{all} contains the union of P^{in} and P^{out}, the converse not necessarily holds true: beside input and output ports, P^{all} may contain also *internal ports*. If a constraint automaton has internal ports, I call it a *composite*; otherwise, I call it a *primitive*.

Definition 18 (states)**.** *A state is an object. $\mathbb{Q}$ denotes the set of all states, ranged over by q. $2^{\mathbb{Q}}$ denotes the set of all sets of states, ranged over by Q.*

Definition 19 (constraint automata)**.** *A constraint automaton is a tuple:*

$$(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0))$$

where:

- $Q \subseteq \mathbb{Q}$ *(states)*
- $(P^{\text{all}}, P^{\text{in}}, P^{\text{out}}) \in 2^{\mathbb{P}} \times 2^{\mathbb{P}} \times 2^{\mathbb{P}}$ *such that:* *(ports)*

$$P^{\text{in}}, P^{\text{out}} \subseteq P^{\text{all}} \ \textbf{and} \ P^{\text{in}} \cap P^{\text{out}} = \emptyset$$

- $M \subseteq \mathbb{M}$ *(memory cells)*
- $\longrightarrow \subseteq Q \times 2^{P^{\text{all}}} \times \mathsf{Good}(P^{\text{all}} \cup {}^\bullet M \cup M^\bullet) \times Q$ *such that:* *(transitions)*

$$\left[\begin{array}{c} q \xrightarrow{P,\phi} q' \ \textbf{implies} \\ \phi \in \mathsf{Good}(P \cup {}^\bullet M \cup M^\bullet) \end{array}\right] \textbf{for all} \ q, q', P, \phi$$

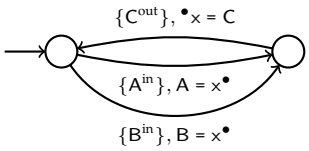
- $(q^0, \mu^0) \in Q \times (M \to \mathbb{D})$ *(initial configuration)*

$\mathbb{A}\text{UTOM}$ *denotes the set of all constraint automata, ranged over by $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}$. $2^{\mathbb{A}\text{UTOM}}$ denotes the set of all sets of constraint automata, ranged over by A, B. $2^{2^{\mathbb{A}\text{UTOM}}}$ denotes the set of all sets of sets of constraint automata, ranged over by $\mathcal{A}$.*

Figure 2.2 shows a first example. In graphical representations of constraint automata, I annotate ports in synchronization constraints with superscripts "in" and "out" to indicate their polarity; internal ports have no explicit annotation. Henceforth, let $\mathsf{Stat}(\boldsymbol{a})$ denote the state space of a constraint automaton $\boldsymbol{a}$, let $\mathsf{Port}(\boldsymbol{a})$ denote its set of ports, let $\mathsf{Input}(\boldsymbol{a})$ and $\mathsf{Output}(\boldsymbol{a})$ denote its set of input and output ports, let $\mathsf{Memor}(\boldsymbol{a})$ denote its set of memory cells, let $\mathsf{Trans}(\boldsymbol{a})$ denote its transition relation, let $\mathsf{init}(\boldsymbol{a})$ denote its initial configuration, and let $\mathsf{Dc}(\boldsymbol{a})$ denote the set of data constraints that occur on its transitions.

$$
\begin{aligned}
Q &= \{q_1, q_2\} \\
(P^{\text{all}}, P^{\text{in}}, P^{\text{out}}) &= (\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}, \{\mathsf{A}, \mathsf{B}\}, \{\mathsf{C}\}) \\
M &= \{\mathsf{x}\} \\
\longrightarrow &= \left\{ \begin{array}{l} (q_1, \{\mathsf{A}\}, \mathsf{A} = \mathsf{x}^\bullet, q_2), \\ (q_1, \{\mathsf{B}\}, \mathsf{B} = \mathsf{x}^\bullet, q_2), \\ (q_2, \{\mathsf{C}\}, {}^\bullet\mathsf{x} = \mathsf{C}, q_1), \end{array} \right\} \\
(q^0, \mu^0) &= (q_1, \{\mathsf{x} \mapsto 0\})
\end{aligned}
$$

Textual representation



Graphical representation

Figure 2.2: Constraint automaton for the LateAsyncMerger2 protocol in the producer/consumer example in Chapter 1. One producer has access to port A, the other producer has access to port B, the consumer has access to port C, and the producers and the consumer use buffer x for temporary storage of data.

"This thesis' constraint automata" (i.e., first-order constraint automata with memory) generalize "original constraint automata" previously developed by Baier et al. [BSAR06]: original constraint automata constitute the subset of this thesis' constraint automata without memory cells and without existential quantification, data functions, and data relations in data constraints. By subsequently removing also data constraints, this thesis' constraint automata further reduce to *port automata*, first studied by Koehler and Clarke [KC09]. (In turn, the semantic domain of the *connector algebras* developed by Bliudze and Sifakis essentially consist of single-state port automata [BS08, BS10].) Extensions of original constraint automata with memory (but still without existential quantification, data functions, and data relations in data constraints) include *constraint automata with state memory*, used in work of Pourvatan et al. and formalized in a categorical setting by Krause et al. [KGdV13, PSAB12, PSHA12]. To my knowledge, Klüppelholz and Baier first articulated the distinction between explicit representations of interaction letters (*concurrent* I/O *operations* in their terminology) and symbolic representations of sets of interaction letters as transition labels [KB09]. Pushing their symbolic representation one step further than here, though, Klüppelholz and Baier in fact combine synchronization constraints and data constraints into single symbolic objects, called I/O *constraints*. As another novelty at that time, their calculus of I/O constraints supports arbitrary data relations (but no data functions or existential quantification). Under some restrictions [Klü12], the model checker developed by Baier et al. can verify original constraint automata extended with data functions and data relations [BBK+10, BBKK09a]. Finally, the encoding of original constraint automata as purely logical constraints developed by Clarke et al. and Proença also supports existential quantification, unary data functions, and data relations, as this thesis' constraint automata [CPLA11, PC13a, PC13b, Pro11].

Behavior, Equivalence, and Congruence

The memory cells in constraint automata may remind one of *stacks* in classical *pushdown automata* [HMU06]: both memory cells and stacks register behaviorally relevant—yet ultimately hidden—information, beyond observable behavior. In defining the *runs* of a constraint automaton, I therefore recall and adopt the following concepts from pushdown automata theory. An *instantaneous description* of a pushdown automaton consists of three elements: its current state, the remaining *input tape*, and the current content of its stack. A pushdown automaton can *move* from one instantaneous description to the next by firing a transition out of its current state, thereby possibly changing its state, certainly consuming the first *input symbol* on the tape (i.e., a letter), and possibly changing its stack. A sequence of successive moves, starting from an initial instantaneous description, results in a run. By replacing "input tape" with "interaction word", "input symbol" with "interaction letter", and "stack" with "set of memory cells", the previous concepts become applicable also to constraint automata. First, I formally define an instantaneous description as a triple (q, w, μ) consisting of a state q, an interaction word w, and a memory snapshot μ.

Definition 20 (instantaneous descriptions). $\mathbb{DESCR} = \mathbb{Q} \times \mathbb{WORD} \times \mathbb{SNAPSH}$ *denotes the set of all instantaneous descriptions.*

For a constraint automaton $\boldsymbol{a}$ with memory cells M to move from an instantaneous description $(q, \lambda w, \mu)$ to another instantaneous description (q', w, μ'), several conditions must hold. Obviously, $\boldsymbol{a}$ should have a transition (q, P, ϕ, q') from state q to state q'. Second, memory snapshots μ and μ' should have exactly M as their domain (i.e., in making a transition, $\boldsymbol{a}$ cannot affect memory cells that it does not know about). Third, interaction letter λ should satisfy the synchronization constraint of the transition: λ should have exactly P as its domain. Finally, the data assignment composed of λ, μ and μ' should satisfy data constraint ϕ. These conditions ensure that at least the first instance of interaction in the chain of interaction modeled by interaction word λw respects the protocol modeled by $\boldsymbol{a}$. Importantly, unless ϕ explicitly states otherwise (e.g., by using the `K` predicate), the content of memory cells in M can nondeterministically change during a move. As an alternative to the previous conditions, $\boldsymbol{a}$ can also move from (q, w, μ) to (q', w, μ') if a transition from q to q' with an empty synchronization constraint exists: such an *unobservable transition* does not contribute to the observable chain of interaction modeled by w.

Definition 21 (moves-to). $\vdash \subseteq \mathbb{AUTOM} \times \mathbb{DESCR} \times \mathbb{DESCR}$ *denotes the smallest relation induced by the rules in Figure 2.3.*

Let w denote an interaction word, and let $\boldsymbol{a}$ denote a constraint automaton with initial configuration (q^0, μ^0). If $\boldsymbol{a}$ has an infinite run starting from instantaneous description (q^0, w, μ^0), interaction word w belongs to the interaction language of $\boldsymbol{a}$. In that case, $\boldsymbol{a}$ accepts w. Because internal choices in a constraint

$$\frac{\begin{array}{l} q \xrightarrow{P,\phi} q' \\ \textbf{and } \mathrm{Dom}(\mu) = \mathrm{Dom}(\mu') = M \\ \textbf{and } \mathrm{Dom}(\lambda) = P \\ \textbf{and } \lambda \cup \{^\bullet m \mapsto \mu(m) \mid m \in M\} \cup \{m^\bullet \mapsto \mu'(m) \mid m \in M\} \models \phi \end{array}}{(q, \lambda w', \mu) \vdash_{(\cdot,\cdot,M,\longrightarrow,\cdot)} (q', w', \mu')} \tag{2.11}$$

$$\frac{\begin{array}{l} q \xrightarrow{\emptyset,\phi} q' \\ \textbf{and } \mathrm{Dom}(\mu) = \mathrm{Dom}(\mu') = M \\ \textbf{and } \{^\bullet m \mapsto \mu(m) \mid m \in M\} \cup \{m^\bullet \mapsto \mu'(m) \mid m \in M\} \models \phi \end{array}}{(q, w, \mu) \vdash_{(\cdot,\cdot,M,\longrightarrow,\cdot)} (q', w, \mu')} \tag{2.12}$$

Figure 2.3: Addendum to Definition 21

automaton do not matter for modeling protocols in this thesis (i.e., I model protocols only in terms of observable data-flows on ports), I consider the set of all interaction words accepted by $\boldsymbol{a}$ the behavior of $\boldsymbol{a}$.

Definition 22 (behavior)**.** $\mathsf{Behav} : \mathbb{A}\text{UTOM} \to \mathbb{L}\text{ANG}$ *denotes the function defined by the following equation:*

$$\mathsf{Behav}(\boldsymbol{a}) = \{w \mid \mathsf{init}(\boldsymbol{a}) = (q, \mu) \textbf{ and } (q, w, \mu) \vdash_{\boldsymbol{a}} (q', w', \mu') \vdash_{\boldsymbol{a}} \cdots\}$$

The existence of an infinite run of a constraint automaton $\boldsymbol{a}$ on an interaction word w essentially means that the protocol modeled by $\boldsymbol{a}$ admits every step of "the way data is exchanged through ports" modeled by w. As such, $\boldsymbol{a}$—and in particular its transition relation—indeed models "a set of rules that control the way data is exchanged through ports", thereby faithfully capturing the dictionary definition on page 27.

Two constraint automata model the same protocol if those two automata have the same behavior, up to internal choices (i.e., they accept exactly the same interaction words). This intuition induces a straightforward and natural notion of *behavioral equivalence* on constraint automata, based on equality of their accepted interaction languages.

Definition 23 (behavioral equivalence)**.** $\approx \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\mathsf{Behav}(\boldsymbol{a}_1) = \mathsf{Behav}(\boldsymbol{a}_2)}{\boldsymbol{a}_1 \approx \boldsymbol{a}_2} \tag{2.13}$$

Proving behavioral equivalence between constraint automata plays an important role in establishing the correctness of protocol optimizations, including those presented later in this thesis. As in Milner's work on CCS [Mil89], however, the previous behavioral equivalence based on interaction languages has

a practical problem: although it *does* denote an equivalence relation in its technical sense, $\approx$ *does not* denote a congruence relation under certain operations (discussed shortly). This complicates proving behavioral equivalences.

Inspired by Milner's *bisimulation*—but no less by the variant of bisimulation developed by Baier et al. [BSAR06]—I therefore introduce a congruence relation on constraint automata that subsumes $\approx$. Then, in the rest of this thesis, to prove behavioral equivalence between constraint automata, instead, I prove *behavioral congruence* to imply behavioral equivalence. Keep in mind, however, that only behavioral equivalence truly matters in the end; behavioral congruence just serves as a means to achieve that end.

First, I define the *behavioral preorder* to establish when a constraint automaton $\boldsymbol{a}_2$ *simulates* a constraint automaton $\boldsymbol{a}_1$. In that case, a relation R on the states of $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ exists such that, for every state q_1 of $\boldsymbol{a}_1$, if:

- R relates q_1 to a state q_2 of $\boldsymbol{a}_2$
- and $\boldsymbol{a}_1$ has a transition from q_1 to a state q_1' that admits a set of interaction letters Λ,

then:

- $\boldsymbol{a}_2$ has a transition from q_2 to a state q_2' that admits *at least* the interaction letters in Λ
- and R relates q_1' to q_2'.

In other words, $\boldsymbol{a}_2$ can always simulate every transition that $\boldsymbol{a}_1$ can make, even its unobservable transitions. One may weaken this notion of strong simulation by ignoring unobservable transitions, but I neither need nor pursue such a notion of weak simulation in this thesis.

Definition 24 (behavioral preorder). $\preceq \subseteq 2^{\mathbb{Q}\times\mathbb{Q}} \times \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\begin{array}{l} R \subseteq Q_1 \times Q_2 \ \textbf{and} \ q_1^0 \ R \ q_2^0 \\ \textbf{and} \quad \left[\begin{array}{c} \left[\left[\begin{array}{l} q_1 \xrightarrow{P,\phi_1}_1 q_1' \\ \textbf{and} \ \ q_1 \ R \ q_2 \end{array}\right] \ \textbf{implies} \ \ \phi_1 \Rightarrow \bigvee \left\{ \phi_2 \left| \begin{array}{l} q_2 \xrightarrow{P,\phi_2}_2 q_2' \\ \textbf{and} \ \ q_1' \ R \ q_2' \end{array} \right. \right\}\right] \\ \textbf{for all} \ \ q_1, q_1', q_2, P, \phi_1 \end{array}\right] \end{array}}{\begin{array}{c} (Q_1, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow_1, (q_1^0, \mu^0)) \\ \preceq_R (Q_2, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow_2, (q_2^0, \mu^0)) \end{array}} \tag{2.14}$$

(Technically, the ternary relation $\preceq$ does not denote a preorder because of its third operand, but I ignore this minor detail here by abuse of terminology.) Out of the behavioral preorder, I construct the behavioral congruence.

Definition 25 (behavioral congruence). $\simeq \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\left[\boldsymbol{a}_1 \preceq_R \boldsymbol{a}_2 \ \textbf{and} \ \boldsymbol{a}_2 \preceq_{R^{-1}} \boldsymbol{a}_1\right] \ \textbf{for some} \ R}{\boldsymbol{a}_1 \simeq \boldsymbol{a}_2} \tag{2.15}$$

The following theorem states that behavioral congruence implies behavioral equivalence. This result shows that I can indeed use $\simeq$ to establish $\approx$.

Theorem 1. $\boldsymbol{a}_1 \simeq \boldsymbol{a}_2$ **implies** $\boldsymbol{a}_1 \approx \boldsymbol{a}_2$

In the next subsection, I present actual congruence results for $\simeq$.

Multiplication and Subtraction

As in all engineering disciplines, *composition*—the act of building more complex objects out of simpler ones—and *abstraction*—the act of hiding objects' irrelevant details—play an important role in software engineering. This holds true also for implementing protocol specifications. Therefore, I define two operations on constraint automata: *multiplication* for composition and *subtraction* for abstraction.

Multiplication consumes two constraint automata $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ as input and produces a constraint automaton as output. I formally define multiplication on constraint automata as a partial function. This partiality models that not all protocols can compose into a new one: two protocols can compose only if (i) each of their *shared ports* serves as an input port in one protocol and as an output port in the other and (ii) these two protocols have no shared buffers. Because constraint automata have a rather involved structure, the formal definition of multiplication may look deceivingly complex. Therefore, I first present a more informal description to explain the main concepts involved. Let $\boldsymbol{a}_1$ denote $(Q_1, (P_1^{\text{all}}, P_1^{\text{in}}, P_1^{\text{out}}), M_1, \longrightarrow_1, (q_1^0, \mu_1^0))$, and let $\boldsymbol{a}_2$ denote $(Q_2, (P_2^{\text{all}}, P_2^{\text{in}}, P_2^{\text{out}}), M_2, \longrightarrow_2, (q_2^0, \mu_2^0))$. Assuming that conditions (i) and (ii) hold true, I take the following steps to multiply $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$.

- First, I take the Cartesian product of Q_1 and Q_2 as the new set of states, and I take the pair of q_1^0 and q_2^0 as the new initial state (inside the new initial configuration).
- Second, I take the union of P_1^{all} and P_2^{all} as the new set of all ports. Subsequently, I put every port in $P_1^{\text{in}} \cup P_2^{\text{in}}$ in the new set of input ports, except those that serve *also* as output ports. Similarly, I put every port in $P_1^{\text{out}} \cup P_2^{\text{out}}$ in the new set of output ports, except those that serve *also* as input ports. Ports with "mixed polarity" (e.g., those that serve as input port in $\boldsymbol{a}_1$ and as output port in $\boldsymbol{a}_2$) become internal ports in the product.
- Third, I take the union of M_1 and M_2 as the new set of memory cells, and I take the union of μ_1^0 and μ_2^0 as the new initial memory snapshot (inside

the new initial configuration). Because (ii) holds true, $\mu_1^0 \cup \mu_2^0$ denotes a well-defined function over domain $M_1 \cup M_2$.

- Finally, I must construct a new transition relation out of $\longrightarrow_1$ and $\longrightarrow_2$. I do so with three rules.

  The first rule states that a transition of $\boldsymbol{a}_1$ involving a shared port can fire iff a transition of $\boldsymbol{a}_2$ involving that same shared port synchronously fires. In other words, $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ must *agree* on synchronously firing transitions involving shared ports. The concept of agreement plays an important role in Chapter 5. Here, I call the kind of agreement required between $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ *weak*. The reason for this particular modifier becomes clear in Chapter 5, where I also introduce a notion of *strong* agreement.

  The second rule states that a transition of $\boldsymbol{a}_1$ involving no shared ports can fire at any time. This means that the protocol modeled by $\boldsymbol{a}_1$ admits the instances of interaction controlled by that transition regardless of the protocol modeled by $\boldsymbol{a}_2$ in the composition of those protocols. Indeed, if the protocol modeled by $\boldsymbol{a}_2$ does not know about a port, it cannot exercise any kind of control over how interaction occurs on that port. (Formally, I should carefully ensure that firings of such a transition in $\boldsymbol{a}_1$ do not affect the memory cells in $\boldsymbol{a}_2$. After all, during every move, the content of memory cells may nondeterministically change unless explicitly stated otherwise.) The third rule states the same as the second rule but with $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ reversed.

For technical convenience—especially later in this thesis—I define multiplication as just explained in three steps. First, I define weak agreement. Second, I define an agreement-parametric multiplication, which takes three instead of two operands: an agreement relation and two constraint automata. Although not directly useful, in Chapter 5, agreement-parametric multiplication enables me to straightforwardly define a different multiplication, based on another form of agreement. Third, I instantiate generalized multiplication with weak agreement.

Definition 26 (agreement)**.** *An agreement relation is a relation $*$ such that:*

$$* \subseteq (2^{\mathbb{P}} \times 2^{\mathbb{P}}) \times (2^{\mathbb{P}} \times 2^{\mathbb{P}}) \textbf{ and } \left[\begin{array}{c} \left[\begin{array}{c} (P_1^{\text{all}}, P_1) * (P_2^{\text{all}}, P_2) \textbf{ implies} \\ \left[\begin{array}{c} (P_2 \setminus P_1) \cap P_1^{\text{all}} = \emptyset \\ \textbf{and } (P_1 \setminus P_2) \cap P_2^{\text{all}} = \emptyset \end{array}\right] \end{array}\right] \\ \textbf{for all } P_1, P_1^{\text{all}}, P_2, P_2^{\text{all}} \end{array}\right]$$

$\mathbb{A}$GREEM *denotes the set of all agreement policies.*

$$\frac{q_1 \xrightarrow{P_1,\phi_1}_1 q_1' \ \textbf{and} \ q_2 \xrightarrow{P_2,\phi_2}_2 q_2' \ \textbf{and} \ (P_1^{\text{all}}, P_1) * (P_2^{\text{all}}, P_2)}{q \xrightarrow{P_1 \cup P_2, \phi_1 \wedge \phi_2}_{\circledast} q'} \tag{2.17}$$

$$\frac{q_1 \xrightarrow{P_1,\phi_1}_1 q_1' \ \textbf{and} \ q_2 \in Q_2 \ \textbf{and} \ P_2^{\text{all}} \cap P_1 = \emptyset}{(q_1, q_2) \xrightarrow{P_1, \phi_1 \wedge \mathtt{K}(M_2)}_{\circledast} (q_1', q_2)} \tag{2.18}$$

$$\frac{q_2 \xrightarrow{P_2,\phi_2}_2 q_2' \ \textbf{and} \ q_1 \in Q_1 \ \textbf{and} \ P_1^{\text{all}} \cap P_2 = \emptyset}{(q_1, q_2) \xrightarrow{P_2, \phi_2 \wedge \mathtt{K}(M_1)}_{\circledast} (q_1, q_2')} \tag{2.19}$$

Figure 2.4: Addendum to Definition 28

Definition 27 (weak agreement). $\Diamond \subseteq (2^{\mathbb{P}} \times 2^{\mathbb{P}}) \times (2^{\mathbb{P}} \times 2^{\mathbb{P}})$ *denotes the smallest relation induced by the following rule:*

$$\frac{P_1 \subseteq P_1^{\text{all}} \ \textbf{and} \ P_2 \subseteq P_2^{\text{all}} \ \textbf{and} \ P_1^{\text{all}} \cap P_2 = P_2^{\text{all}} \cap P_1}{(P_1^{\text{all}}, P_1) \Diamond (P_2^{\text{all}}, P_2)} \tag{2.16}$$

Lemma 1. $\Diamond \in \mathbb{A}\text{GREEM}$

Definition 28 (agreement-parametric multiplication)**.**
$\circledast : \mathbb{A}\text{GREEM} \times \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM} \rightharpoonup \mathbb{A}\text{UTOM}$ *denotes the partial function defined by the following equation:*

$$\begin{pmatrix} Q_1, \\ \begin{pmatrix} P_1^{\text{all}}, \\ P_1^{\text{in}}, \\ P_1^{\text{out}} \end{pmatrix}, \\ M_1, \\ \longrightarrow_1, \\ (q_1^0, \mu_1^0) \end{pmatrix} \circledast_* \begin{pmatrix} Q_2, \\ \begin{pmatrix} P_2^{\text{all}}, \\ P_2^{\text{in}}, \\ P_2^{\text{out}} \end{pmatrix}, \\ M_2, \\ \longrightarrow_2, \\ (q_2^0, \mu_2^0) \end{pmatrix} = \begin{pmatrix} Q_1 \times Q_2, \\ P_1^{\text{all}} \cup P_2^{\text{all}}, \\ \begin{pmatrix} (P_1^{\text{in}} \cup P_2^{\text{in}}) \setminus (P_1^{\text{out}} \cup P_2^{\text{out}}), \\ (P_1^{\text{out}} \cup P_2^{\text{out}}) \setminus (P_1^{\text{in}} \cup P_2^{\text{in}}) \end{pmatrix}, \\ M_1 \cup M_2, \\ \longrightarrow_{\circledast}, \\ ((q_1^0, q_2^0), \mu_1^0 \cup \mu_2^0) \end{pmatrix}$$

$$\textbf{if} \begin{bmatrix} P_1^{\text{all}} \cap P_2^{\text{all}} = \\ \quad (P_1^{\text{in}} \cap P_2^{\text{out}}) \cup (P_1^{\text{out}} \cap P_2^{\text{in}}) \\ \textbf{and} \ \ M_1 \cap M_2 = \emptyset \end{bmatrix}$$

where $\longrightarrow_{\circledast}$ *denotes the smallest relation induced by the rules in Figure 2.4.*

Definition 29 (multiplication)**.** $\otimes : \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM} \rightharpoonup \mathbb{A}\text{UTOM}$ *denotes the partial function defined by the following equation:*

$$\boldsymbol{a}_1 \otimes \boldsymbol{a}_2 = \boldsymbol{a}_1 \circledast_{\Diamond} \boldsymbol{a}_2$$

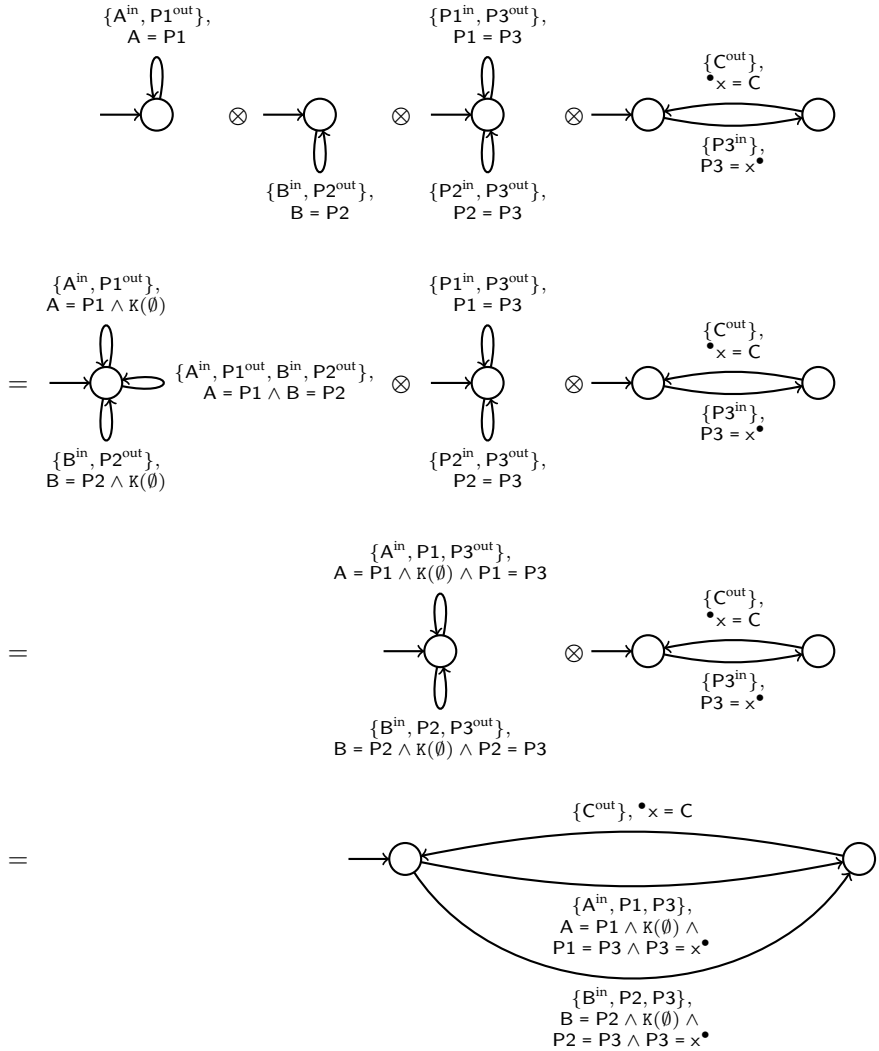


Figure 2.5: Multiplication of four constraint automata

Figure 2.5 shows an example. Henceforth, whenever I write "multiplication", I always mean multiplication as in Definition 29 unless explicitly stated otherwise. Multiplication satisfies commutativity and associativity up-to behavioral congruence.

Essentially, multiplication glues together the constituent protocols modeled by its multiplicands on their shared ports. In particular, every rule involving a shared port in one constituent protocol (i.e., a transition in a constraint automaton) must "synchronize" with every rule involving the same shared port in the other constituent protocol. Such synchronization ensures that every in-

stance of interaction admitted by their composition abides by every relevant rule in *both* constituent protocols. This kind of composition has two interesting properties: *multiparty synchronization* and *indirect synchronization*. Multiparty synchronization means that through successive applications, $\otimes$ can synchronize transitions in one constraint automaton with transitions in multiple other constraint automata. For instance, $\otimes$ synchronizes the transitions in the middle constraint automaton on the second line in Figure 2.5 with transitions in both the left constraint automaton (multiplied on the third line) and the right constraint automaton (multiplied on the fourth line). Indirect synchronization means that through successive applications, $\otimes$ can synchronize transitions in a constraint automaton with transitions in another constraint automaton via a number of "intermediate" constraint automata. For instance, $\otimes$ synchronizes the transitions in the left constraint automaton on the second line in Figure 2.5 with the lower transition in the right constraint automaton (multiplied on the fourth line) via the transitions in the middle constraint automaton (multiplied on the third line). Indirect synchronization enables compositional construction of globally synchronous composites out of locally synchronous primitives.

The following theorems state that $\simeq$ denotes a congruence under $\circledast$ and $\otimes$.

Theorem 2. $\begin{bmatrix} \boldsymbol{a}_1 \circledast_* \boldsymbol{a}_3, \boldsymbol{a}_2 \circledast_* \boldsymbol{a}_4 \in \mathbb{A}\text{UTOM} \\ \textbf{and } \boldsymbol{a}_1 \simeq \boldsymbol{a}_2 \textbf{ and } \boldsymbol{a}_3 \simeq \boldsymbol{a}_4 \end{bmatrix}$ **implies** $\boldsymbol{a}_1 \circledast_* \boldsymbol{a}_3 \simeq \boldsymbol{a}_2 \circledast_* \boldsymbol{a}_4$

Theorem 3. $\begin{bmatrix} \boldsymbol{a}_1 \otimes \boldsymbol{a}_3, \boldsymbol{a}_2 \otimes \boldsymbol{a}_4 \in \mathbb{A}\text{UTOM} \\ \textbf{and } \boldsymbol{a}_1 \simeq \boldsymbol{a}_2 \textbf{ and } \boldsymbol{a}_3 \simeq \boldsymbol{a}_4 \end{bmatrix}$ **implies** $\boldsymbol{a}_1 \otimes \boldsymbol{a}_3 \simeq \boldsymbol{a}_2 \otimes \boldsymbol{a}_4$

Subtraction consumes a constraint automaton $\boldsymbol{a}$ and a port p as input and produces a constraint automaton as output. To subtract p from $\boldsymbol{a}$, I remove p from every set of ports that $\boldsymbol{a}$ consists of, including synchronization constraints of transitions, and I existentially quantify p away in every data constraint.

Definition 30 (subtraction)**.** $\ominus : \mathbb{A}\text{UTOM} \times \mathbb{P} \to \mathbb{A}\text{UTOM}$ *denotes the function defined by the following equation:*

$$(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0)) \ominus p = \\ (Q, (P^{\text{all}} \setminus \{p\}, P^{\text{in}} \setminus \{p\}, P^{\text{out}} \setminus \{p\}), M, \longrightarrow_{\ominus}, (q^0, \mu^0))$$

where $\longrightarrow_{\ominus}$ *denotes the smallest relation induced by the following rule:*

$$\frac{q \xrightarrow{P,\phi} q'}{q \xrightarrow{P \setminus \{p\}, \exists p.\phi}_{\ominus} q'} \tag{2.20}$$

Figure 2.6 shows an example. (The constraint automaton in Figure 2.6 accepts exactly the same interaction language as the interaction language accepted by the constraint automaton in Figure 2.2, modulo the subtracted ports.)

Subtraction as defined in Definition 30 specializes a more general subtraction on constraint automata, where also the right-hand side denotes a con-

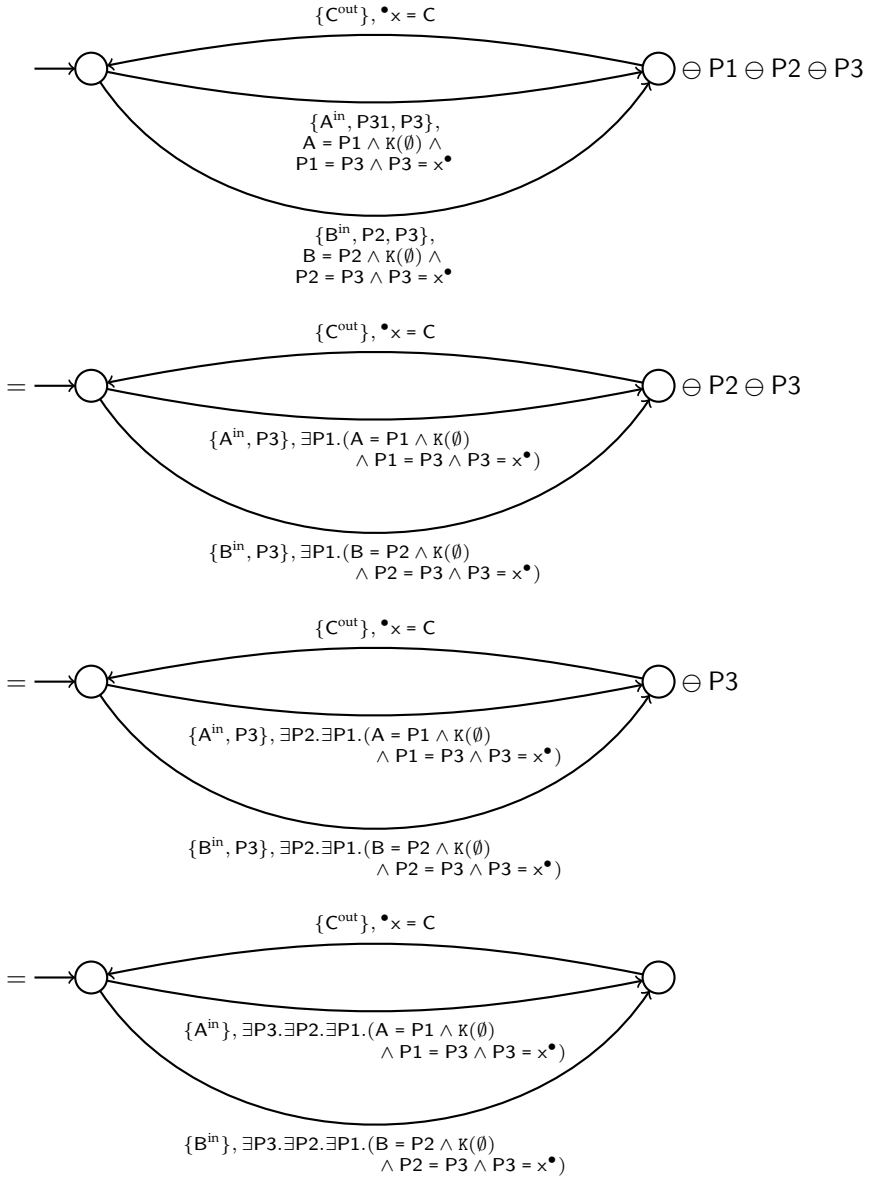


Figure 2.6: Subtraction of ports from the constraint automaton in Figure 2.5

straint automaton (instead of just a port). Because I do not need such general subtraction in this thesis, I skip it here and present only Definition 30.

The following theorem states that $\simeq$ denotes a congruence under $\ominus$.

Theorem 4. $\left[p \in \mathbb{P} \textbf{ and } \boldsymbol{a}_1 \simeq \boldsymbol{a}_2\right] \textbf{ implies } \boldsymbol{a}_1 \ominus p \simeq \boldsymbol{a}_2 \ominus p$

The following theorem states that subtraction *of unshared ports* distributes over multiplication up-to behavioral congruence. Consequently, I can incrementally subtract internal ports, which constraint automata cannot share by definition, during a bigger multiplication.

Theorem 5.

$$\begin{bmatrix} p \notin \mathsf{Port}(\boldsymbol{a}_1) \cap \mathsf{Port}(\boldsymbol{a}_2) \\ \textbf{and } \boldsymbol{a}_1 \otimes \boldsymbol{a}_2 \in \mathbb{A}\text{UTOM} \end{bmatrix} \textbf{ implies } (\boldsymbol{a}_1 \otimes \boldsymbol{a}_2) \ominus p \simeq (\boldsymbol{a}_1 \ominus p) \otimes (\boldsymbol{a}_2 \ominus p)$$

Definition 29 of $\otimes$ extends the definition of multiplication on original constraint automata developed by Baier et al. [BSAR06], mainly by accounting for memory cells in Rules 2.18 and 2.19. Pourvatan et al. only informally define a multiplication on constraint automata with state memory, without mentioning a K-like predicate to account for nondeterministic changes to the content of memory cells [PSAB12, PSHA12]. Krause et al. generalize the multiplication of Pourvatan et al. as pullbacks in a category of constraint automata with state memory [KGdV13], allowing those automata to synchronize not only on ports but also on memory cells and states. The definition of Krause et al. generalizes also Definition 29, but I do not need this level of generality in this thesis. Finally, Klüppelholz introduces another generalization of the multiplication of Baier et al. that, as Definition 29, takes into account ports' direction [Klü12].

Definition 30 of $\ominus$ significantly differs from the subtraction developed by Baier et al. [BSAR06], which not only removes ports but also eliminates unobservable transitions. Klüppelholz calls the former kind of subtraction *structure-preserving* and the latter kind *aggregating* [Klü12]. Memory cells in constraint automata make elegantly defining aggregating subtraction quite challenging; I leave this for future work.

2.2 Practice

(I have not yet submitted the material in this section for publication.)

I developed a Java library for constraint automata, multiplication, and subtraction, with separate classes for constraint automata (class `Automaton`), their states (class `State`), their transitions (class `Transition`), data constraints (class `Constraint`), ports (class `Port`), and more. Importantly, I use these classes only for *representing* constraint automata and not for running them. In particular, class `Port` does not implement interfaces `InputPort` and `OutputPort` in Figure 1.9; I come to that later, in Chapter 4.

To multiply constraint automata, the library often needs to evaluate the weak agreement relation in Definition 27 (i.e., $k_1 k_2$ times, where k_1 and k_2 denote the number of transitions in the multiplicands). To do this efficiently, in terms of both time and space, I wrote a special data structure for sets. Every

time the library constructs a new `Port`, it gives this `Port` a unique positive integer id. Subsequently, it can represent set membership of `Ports` as a binary string, where a `1` in the i-th position means that a set includes the `Port` with id i (similarly, a `0` means exclusion). The library can subsequently store such binary strings as arrays of `int`s (whose length depends on the total number of `Ports` in the two multiplicands) and perform operations on sets—containment, union, intersection, complementation, difference–using integer arithmetic. This special data structure for sets led to significantly better performance than, for instance, `java.util.HashSet`, which I used in earlier versions of this library.

Software engineers using the library can construct `Automatons` either directly or indirectly. In the direct method, software engineers directly use the constructor of `Automaton` (through a factory design pattern) to obtain an empty `Automaton`. Subsequently, because they have an `Automaton` object at their disposal, those software engineers can directly use methods of that object for adding `States` and `Transitions`. However, because the direct method exposes `Automaton` objects to software engineers, those software engineers can use *all* public methods of those objects. For software engineers who work on the library itself (i.e., me), the risks involved seem controllable and reasonably within those software engineers' field of responsibility. For software engineers who merely use the library, in contrast, exposing `Automaton` objects seems a bad idea. After all, those objects have public methods that only the library itself should invoke. Such software engineers, therefore, need a more controlled environment in which they can construct `Automatons`.

In the indirect method, software engineers write a subclass of abstract class `UserDefinedAutomaton`. This abstract class has an `Automaton` among its private fields (inaccessible from its subclasses) and, through its protected methods (accessible from its subclasses), exposes only methods for adding `States` and `Transitions`. The library sets the private `Automaton` in a `UserDefinedAutomaton` through a package-visible method (i.e., `Automaton` and `UserDefinedAutomaton` must live in the same package for this to work). This indirect method allows software engineers who merely use the library to construct their own `Automatons` for subsequent multiplication and subtraction in a safe manner.

The library for constraint automata forms a crucial component of the compiler that I present in Chapters 4–8.

Chapter 3

DSL for Interaction II: Syntax

By their definition in Chapter 2, every constraint automaton models "a set of rules that controls the way data is exchanged through ports". Thus, by the dictionary definition on page 27, every constraint automaton models a protocol. In principle, then, constraint automata per se constitute an intention-expressing DSL for interaction that software engineers can use for implementing their protocol specifications. Exposing software engineers directly to constraint automata, however, has at least one major disadvantage: constraint automata quickly grow prohibitively large.

In this chapter, I present two complementary syntaxes for representing multiplication expressions of constraint automata: an existing graphical syntax based on *Reo* [Arb04] and a new textual syntax called *First-Order Constraint Automata with Memory Language* (FOCAML). The graphical syntax perhaps appeals better to intuition, while the textual syntax has more expressive power. In Section 3.1, I first elaborate on the previously stated disadvantage of using constraint automata directly as syntax. Subsequently, I present Reo and FOCAML. In Section 3.2, I present an editor for FOCAML, including an interpreter, and discuss some nontrivial examples.

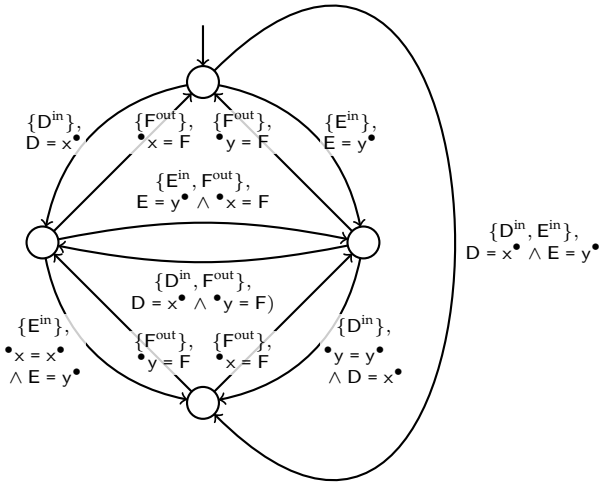


Figure 3.1: Constraint automaton for the EarlyAsyncMerger2 protocol. One producer has access to port D, the other producer has access to port E, the consumer has access to port F, and the producers and the consumer use buffers x and y for temporary storage of data.

3.1 Theory

(I have not yet submitted the material in this section for publication.)

Compositional Construction of Constraint Automata

To illustrate the previous point that constraint automata quickly grow prohibitively large, suppose that I must write a program that consists of k producers and a consumer. My protocol specification states that the producers send their data to the consumer asynchronously, reliably, unordered, but *not* transactionally (cf. the LateAsyncMerger2 protocol in Chapter 1). Depending on the value of k, I call this protocol EarlyAsyncMerger2, EarlyAsyncMerger3, etc. For instance, Figure 3.1 shows a reasonably small constraint automaton for EarlyAsyncMerger2. Generally, however, the constraint automaton for k producers has as many as 2^k states. This example, then, shows that the approach of using constraint automata directly as syntax scales poorly. Instead, software engineers should leverage constraint automata's *compositionality*: they should implement complex protocol specifications out of implementations of simpler ones, by multiplying primitive constraint automata into composites.

Compositional construction forms the core of both the graphical Reo syntax and the textual FOCAML syntax, presented in detail in the next subsections. To implement protocol specifications using these syntaxes, software engineers draw/write declarative multiplication expressions over a *core set* of constraint

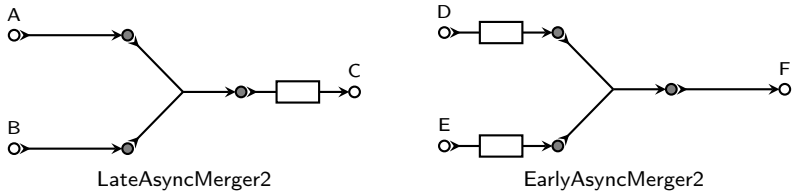


Figure 3.2: Graphical syntax for the constraint automata in Figures 2.2 and 3.1

```
1  LateAsyncMerger2(a,b;c) = {
2    Sync(a;P1) mult Sync(b;P2) mult Merger2(P1,P2;P3) mult Fifo(P3;c)
3  }

4  EarlyAsyncMerger2(d,e;f) = {
5    Fifo(d;P1) mult Fifo(e;P2) mult Merger2(P1,P2;P3) mult Sync(P3;f)
6  }
```

Figure 3.3: Textual syntax for the constraint automata in Figures 2.2 and 3.1

automata. As a preview, Figures 3.2 and 3.3 show the graphical and textual syntax for LateAsyncMerger2 and EarlyAsyncMerger2. As I demonstrate shortly, contrasting their constraint automata, the graphs for EarlyAsyncMerger$k$ grow only linearly in k—instead of exponentially—while its texts even stay constant (one of the reasons why I consider the latter syntax the more expressive one).

Figure 3.4 shows the *parametric constraint automata* that I selected for inclusion in the core set in this thesis, and which essentially mirror the "typical set" of primitives used in Reo (discussed in more detail shortly). Parametric constraint automata define sets of constraint automata, called *families*, whose elements I call *members*. Formally, I define families as functions from the following function space:

$$\bigcup\{\underbrace{\mathbb{N}^i}_{\substack{\text{natural number}\\ \text{parameters}}} \times \underbrace{(\mathbb{D} \cup \mathbb{F} \cup \mathbb{R})^j}_{\substack{\text{extralogical}\\ \text{parameters}}} \times \underbrace{\mathbb{P}^k \times \mathbb{M}^l}_{\substack{\text{unobservable}\\ \text{parameters}}} \times \underbrace{\mathbb{P}^m \times \mathbb{P}^n}_{\substack{\text{observable}\\ \text{parameters}}} \to \mathbb{A}\text{UTOM} \mid i, j, k, l, m, n \geq 0\}$$

Thus, every family has i *natural number parameters*, j *extralogical parameters* for data, data functions, and data relations (i.e., the extralogical elements of the data constraint calculus in Chapter 2), $k + l$ *unobservable parameters* for k internal ports and l memory cells, and $m + n$ *observable parameters* for n input ports and m output ports (where "input" and "output" qualify ports from the protocol perspective). For instance, in Figure 3.4, Sync has two observable parameters, one for an input port and one for an output port, and no other parameters (i.e., $i = j = k = l = 0$ and $m = n = 1$); Fifo has, additionally, an unobservable parameter for a memory cell (i.e., $i = j = k = 0$ and $l = m = n = 1$); BinRel has one extralogical parameter for a data relation and two observable parameters for input ports (i.e., $i = k = l = n = 0$, $j = 1$, and $m = 2$). By instantiating

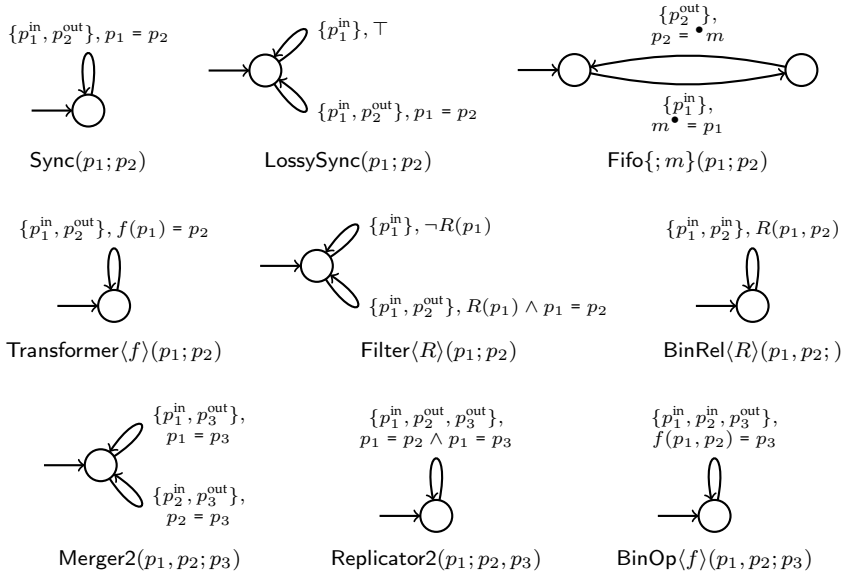


Figure 3.4: Parametric constraint automata in the core set

the parameters of a family with values (i.e., by applying the function to those values), I obtain one of its members. Henceforth, let $\mathbb{F}\text{AM}$ denote the set of all families (i.e., the previous function space).

I use the following notational format for *signatures* of families:

$$\begin{aligned} &\mathit{name}_{\mathit{list\ of\ natural\ number\ parameters}}\langle \mathit{list\ of\ extralogical\ parameters}\rangle \\ &\qquad\qquad \{\mathit{list\ of\ unobservable\ parameters}\} \\ &\qquad\qquad\quad (\mathit{list\ of\ observable\ parameters}) \end{aligned}$$

In this format, as in Figure 3.4, I separate internal ports from memory cells in the list of unobservable parameters by a semicolon, and I do the same for separating input ports from output ports in the list of observable parameters. Also, for notational convenience, I omit lists of natural number, extralogical, and unobservable parameters whenever $i = 0$, $j = 0$, or $k+l = 0$. If a family has natural number parameters, I sometimes write its name with a natural number subscript to denote the "subfamily" corresponding to that natural number, as a kind of function restriction. None of the families defined in Figure 3.4 have natural number parameters. Henceforth, as before, I write names of families in capitalized lower case sans-serif (e.g., Filter, LateAsyncMerger2) and use these names also for their corresponding protocols.

Figure 3.5 describes the behavior of the members of the families defined in Figure 3.4, in terms of the data-flows between ports permitted by those members. Although the choice of core set may matter for software engineers from a usability/productivity perspective, it generally does not matter from a compilation point of view: the techniques that I present later in this thesis work with

Sync$(p_1; p_2)$	Infinitely often atomically $[$accepts a datum d on its input port p_1, then offers d on its output port $p_2]$.
LossySync$(p_1; p_2)$	Infinitely often atomically $[$accepts a datum d on its input port p_1, then either offers d on its output port p_2 or loses $d]$.
Fifo$\{; m\}(p_1; p_2)$	Infinitely often first atomically $[$accepts a datum d on its input port p_1, then stores d in its memory cell $m]$ and subsequently atomically $[$loads d from m, then offers d on its output port $p_2]$.
Filter$\langle R\rangle(p_1; p_2)$	Infinitely often either atomically $[$accepts a datum d on its input port p_1, then establishes that d satisfies data relation R, then offers d on its output port $p_2]$ or atomically $[$accepts a datum d on p_1, then establishes that d violates R, then loses $d]$.
Transformer$\langle f\rangle(p_1; p_2)$	Infinitely often atomically $[$accepts a datum d on its input port p_1, then applies data function f to d, then offers $f(d)$ on its output port $p_2]$.
BinRel$\langle R\rangle(p_1, p_2;)$	Infinitely often, atomically $[$accepts data d_1 and d_2 on its input ports p_1 and p_2, then loses d_1 and $d_2]$, if d_1 and d_2 satisfy R.
Merger2$(p_1, p_2; p_3)$	Infinitely often atomically $[$accepts a datum d either on its input port p_1 or on its input port p_2, then offers d on its output port $p_3]$.
Replicator2$(p_1; p_2, p_3)$	Infinitely often atomically $[$accepts a datum d on its input port p_1, then offers d on its output ports p_2 and $p_3]$.
BinOp$\langle f\rangle(p_1, p_2; p_3)$	Infinitely often atomically $[$accepts data d_1 and d_2 on its input ports p_1 and p_2, then applies data function f to d_1 and d_2, then offers $f(d_1, d_2)$ on its output port $p_3]$.

Figure 3.5: Data-flow description of the behavior of the members of the families defined in Figure 3.4

arbitrary constraint automata and do not depend on my choice of core set—if someone else prefers a different core set than the one in Figure 3.4, no problem.

Graphical Representation: Reo

Instead of exposing software engineers directly to constraint automata—as if constraint automata by themselves constitute a DSL for interaction—software engineers should compositionally construct constraint automata by multiplying members of the families defined in Figure 3.4. To productively develop

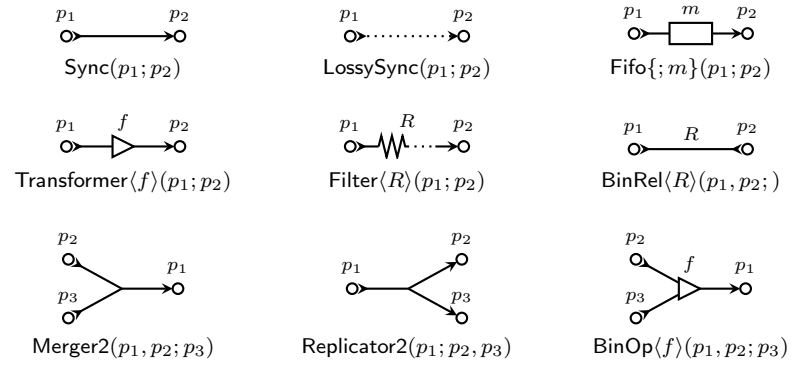


Figure 3.6: Hyperarcs for the families defined in Figure 3.4

the multiplication expressions required for applying this approach, however, software engineers need a more intuitive syntax than just primitive constraint automata. In this subsection, I present a first candidate, based on the previous data-flow description of behavior in Figure 3.5. Using this syntax, already briefly exemplified in Figure 3.2, software engineers draw multiplication expressions as *data-flow hypergraphs*. Every vertex in such a graph represents a port; every hyperarc represents a constraint automaton that controls the interaction on its connected ports. To understand the behavior of a multiplication expression, software engineers can simply "follow the data-flows" through its corresponding hypergraph. Technically, every hypergraph represents a family of constraint automata. To get a member of the family represented by a hypergraph, first, for every hyperarc in that hypergraph, get a member of the family represented by that hyperarc. Subsequently, multiply those per-hyperarc members and subtract all internal ports to get the required constraint automaton.

Figure 3.6 shows a hyperarc for every family defined in Figure 3.4; Figure 3.7 shows, for three new families, hypergraphs composed out of the hyperarcs in Figure 3.6, their *shorthands* (as a single hyperarc), and their parametric constraint automata with simplified data constraints (e.g., I replaced data relation `True` with $\top$ and eliminated a number of existential quantifiers resulting from subtracting internal ports after multiplication). Here and henceforth, white vertices represent input and output ports, while shaded vertices represent internal ports. Members of the three families in Figure 3.6 reappear later in this thesis and behave as follows. Members of SyncDrain simply instantiate BinRel with data relation `True`. Stipulating that `True` holds true of any two data, SyncDrain$(p_1, p_2;)$ infinitely often atomically $\big[$accepts data d_1 and d_2 on its input ports p_1 and p_2, then loses d_1 and $d_2\big]$. AsyncDrain forms the asynchronous version of SyncDrain: AsyncDrain$(p_1, p_2;)$ infinitely often atomically $\big[$accepts a datum d either on its input port p_1 or on its input port p_2, then loses $d\big]$. Finally, Blocker forms a reliable, strict version of Filter: Blocker$\langle R \rangle(p_1; p_2)$ infinitely often $\big[$accepts a datum d on its input port p_1, then offers d on its output port p_2,

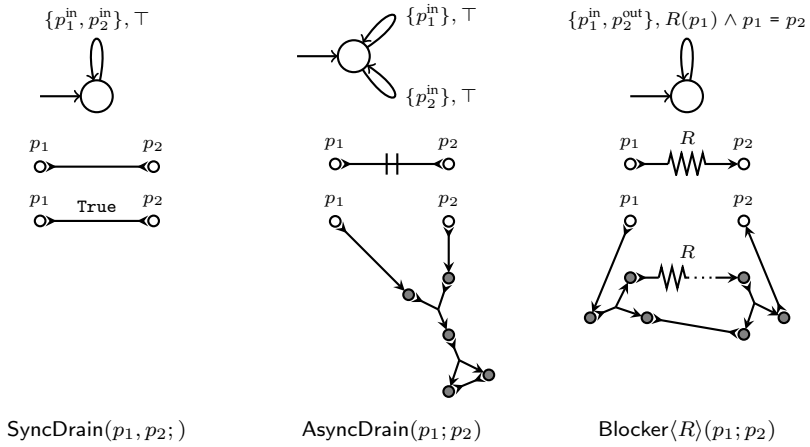


Figure 3.7: Hypergraphs, shorthands, and parametric constraint automata for families SyncDrain, AsyncDrain, and Blocker

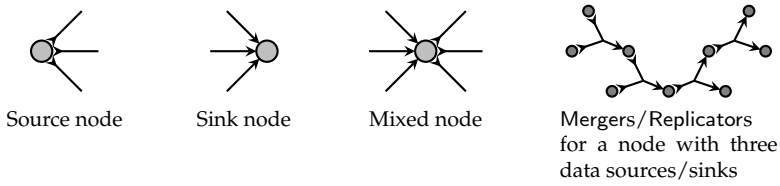


Figure 3.8: Nodes

if d satisfies data relation R].

The previous graphical syntax essentially yields Reo [Arb04, Arb11], an existing graphical language for compositional construction of interaction protocols, manifested as *circuits*. Circuits consist of typed *channels* (edges) and *nodes* (vertices), organized in a graph-like structure. The type of a channel determines both its data-flow behavior and the appearance of its corresponding edge. Every channel consists of two *ends* and a constraint that relates the timing and the contents of the data-flows at those ends. Channel ends have one of two types: *source ends* accept data into their channels (i.e., a source end of a channel connects to that channel's data source/producer), while *sink ends* dispense data out of their channels (i.e., a sink end of a channel connects to that channel's data sink/consumer). Reo makes no other assumptions about channels and allows, for instance, channels with two source ends. Every family defined in Figure 3.4 with two ports corresponds to a channel type in Reo; the first two rows in Figure 3.6 show their corresponding edges. Of these six channel types, only BinRel does not yet appear in the literature on Reo; the other channel types have roughly the same appearance in this thesis as in that literature. Users of Reo may freely define their own channels with custom semantics.

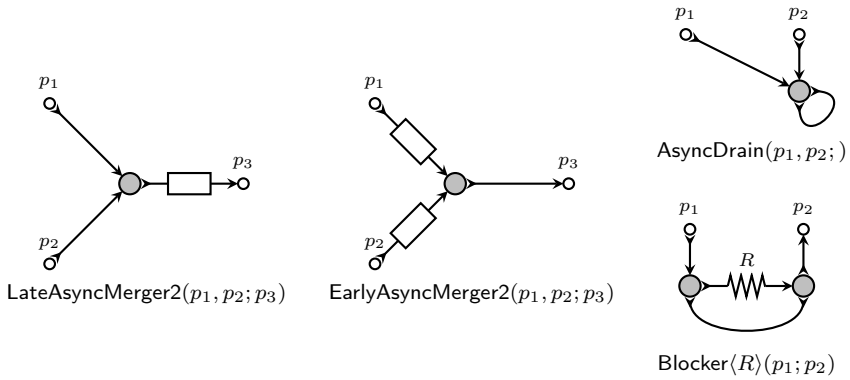


Figure 3.9: Circuits for families LateAsyncMerger2, EarlyAsyncMerger2, AsyncDrain, and Blocker (cf. Figures 3.2 and 3.7)

Channel ends coincide on nodes. Every node has at least one coincident channel end. Depending on its coincident channel ends, a node has one of the three types shown in Figure 3.8. A *source node* has only coincident source ends. Similarly, a *sink node* has only coincident sink ends. Finally, a *mixed node* has both coincident source and coincident sink ends. The source nodes and sink nodes of a circuit constitute its set of *boundary nodes*. The boundary nodes of a circuit permit I/O operations, while a circuit uses its mixed nodes only for internally routing data. Every sink channel end coincident on a node serves as a data source for that node. Analogously, every source channel end coincident on a node serves as a data sink for that node.

Contrasting channels, all nodes have the same, fixed data-flow behavior: repeatedly, a node nondeterministically selects an available datum out of one of its data sources and replicates this datum into each of its data sinks. A node's nondeterministic selection and its subsequent replication constitute one atomic execution step; nodes cannot store, generate, or lose data. Members of the Merger2 family model the nondeterministic selection behavior of a node with two data sources. Similarly, members of the Replicator2 family model the replication behavior of a node with two data sinks. A node with m data sources and n data sinks then corresponds to the multiplication of $m-1$ Merger2 members and $n-1$ Replicator2 members. Figure 3.8 exemplifies this for $m = n = 3$. Henceforth, because it makes the previous data-flow hypergraphs more concise, instead of explicitly drawing internal sequences of Merger2 and Replicator2 members, I collapse them into nodes. Figure 3.9 exemplifies this for previous hypergraphs. In figures, nodes have a larger diameter than ports (twice as large, in fact) and a slightly lighter shade of gray than internal ports. By borrowing this node notation from Reo, I essentially adopt Reo as a graphical syntax for multiplication expressions of constraint automata. Therefore, as in Reo, I call the previous hypergraphs just circuits in the rest of this thesis.

As another example, Figure 3.10 shows the, by now, classical circuit for

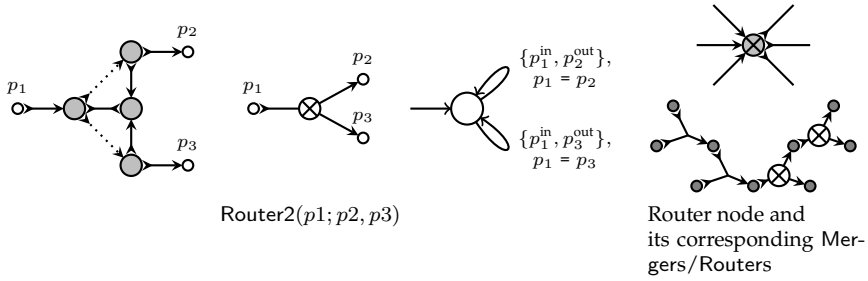


Figure 3.10: Circuit, shorthand, and parametric constraint automaton for family Router2 (left) and router node (right)

the Router2 family. Router2 forms the "inverse" of Merger2: Router2$(p_1; p_2, p_3)$ infinitely often atomically $\big[$accepts a datum d on its input port p_1, then offers d either on its output port p_2 or on its output port $p_3\big]$. The inverse property means that Merger2$(p_1; p_2, p_3)$ and Router2$(p_2, p_3; p_1)$ have exactly the same transitions; just the directions of their ports differs. In Reo, Router2 has so many applications, as a building block for more complex circuits, that it gave rise to its own special node type: *router nodes*. Merely a syntactic sugar (i.e., a graphical shorthand for its equivalent pure subcircuit), a router node has nondeterministic selection behavior not only for its data sources but also for its data sinks. A router node with m data sources and n data sinks then corresponds to the multiplication of $m - 1$ Merger2 members and $n - 1$ Router2 members, as shown in Figure 3.10. As with sequences of Merger2 and Replicator2 members, I collapse sequences of Merger2 and Router2 members into router nodes.

For a circuit to make a global execution step—usually instigated by pending I/O-operations on its boundary nodes—its channels and its nodes must reach consensus about their global behavior, to guarantee mutual consistency of their local execution steps (e.g., a node should not replicate a data item into a channel with an already full buffer). Then, circuit-wide data-flow emerges.

Arbab originally introduced Reo for coordinating components in component-based systems [Arb04]. Since its introduction, however, researchers have used Reo also in other contexts where concurrency and interaction play a role, including service-oriented systems, multi-agent systems and even biological systems (Arbab provides references [Arb11]); to adopt Reo in the context of multicore processors seems only a natural next step in the evolution of Reo's application domains. By now, many formal semantics of Reo exist [**J**A12]. In fact, Baier et al. originally presented constraint automata as a semantics for Reo [BSAR06]. I took the opposite approach in this thesis: I started from constraint automata as an intention-expressing mechanism for modeling protocols and now adopt Reo as a first possible syntax. Other options include zero-safe Petri nets, BPMN, BPEL, and UML activity/sequence diagrams, each of whose constructs easily translate into constraint automata [AKM08, AM08, CKA10, MAB11, TVMS08]. Alternatively, the *connector algebras* of Bliudze and Sifakis

may serve as a syntax for (at least a subset of) constraint automata [BS08, BS10]. Finally, Dokter et al. recently compared Reo and BIP [D**J**AB15].

Textual Representation: FOCAML

Although wonderful for quickly scribbling circuits, visualizing data-flows, and prototyping, Reo's graphical syntax has a disadvantage: it does not support repetition or instantiation constructs. For instance, in the beginning of this section, I introduced EarlyAsyncMerger2, EarlyAsyncMerger3, etc., as separate families of constraint automata. It makes more sense, however, to define only one family LateAsyncMerger with a natural number parameter k that controls the number of input ports of its members. Reo does not support such natural number parametrization, forcing me to draw a separate circuit for every possible number of producers. To overcome this limitation, I present a second syntax for representing multiplication expressions of constraint automata. As stated in the introduction of this chapter, I call this textual syntax FOCAML. The full *concrete syntax* of FOCAML appears in a separate technical report [**Jon**16]; below, I present a summary sufficient for understanding the examples in this thesis.

Every FOCAML program consists of a number of nonrecursive *family definitions* (i.e., protocol subprograms) and a *main definition* (i.e., a main subprogram).

- A family definition consists of a *signature* and a *body*.

  A signature declares the formal parameters of a family definition and has a structure very similar to the function signatures of families used so far: every signature consists of a name, an optional list of natural number parameters between square brackets, an optional list of extralogical parameters between angle brackets, and a mandatory list of port and/or *port array* parameters between parentheses (with inputs and outputs separated by a semicolon). Because of these bracketing conventions, natural number parameters, extralogical parameters, and port parameters require no additional type annotations; array parameters, in contrast, have a "`[]`" suffix. Figures 3.3, 3.11, and 3.12 show examples of signatures (ignore their bodies for the moment). Contrasting the function signatures of families used so far, signatures in FOCAML have no unobservable parameters. Instead, a FOCAML interpreter automatically generates fresh internal ports and memory cells by need.

  Arrays allow software engineers to pass multiple ports to the body of a family definition through a single parameter. Technically, every array parameter in a signature also implicitly declares an extra natural number parameter for its length, accessible through the `#` operator (as exemplified in the body of `Merger` and `Router` in Figure 3.12). This implicit kind of natural number parametrization enables software engineers to write, for instance, `Merger` (which generalizes Merger2 from having two to k input ports, through an array parameter). Alternatively, the explicit kind of

```
1  SyncDrain(in1,in2;) = { BinRel<'True'>(in1,in2;) }

2  AsyncDrain(in1,in2;) = {
3    Sync(in1;P1)
4    mult Sync(in2;P2)
5      mult Merger2(P1,P2;P3)
6        mult Replicator2(P3;P4,P5)
7          mult SyncDrain(P4,P5;)
8  }

9  Blocker<R>(in;out) = {
10   Sync(in;P1)
11     mult Replicator2(P1;P2,P6)
12       mult Filter<R>(P2;P3)
13         mult Replicator2(P3;P4,P5)
14           mult Sync(P4;out)
15           mult SyncDrain(P5,P6;)
16 }

17 Router2(in;out1,out2) = {
18   Sync(in;P1)
19     mult Replicator2(P1;P2,P14)
20       mult Replicator2(P2;P3,P8)
21         mult LossySync(P3;P4)
22           mult Replicator2(P4;P5,P6)
23             mult Sync(P5;out1)
24             mult Sync(P6;P7)
25         mult LossySync(P8;P9)
26           mult Replicator2(P9;P10,P11)
27             mult Sync(P10;out2)
28             mult Sync(P11;P12)
29               mult Merger2(P7,P12;P13)
30                 mult SyncDrain(P13,P14;)
31 }
```

Figure 3.11: FOCAML definitions for families SyncDrain, AsyncDrain, Blocker, and Router

natural number parametrization, between square brackets in a signature, enables software engineers to write, for instance, `FifoK` (which generalizes Fifo from having a 1-capacity to a k-capacity buffer). FOCAML's *abstract syntax*, presented shortly, contains the available array constructors (see also the concrete syntax [**Jon**16]); its *denotational semantics*, also presented shortly, defines their straightforward meaning.

The body of a family definition consists of an expression over instantiated signatures, operator **mult**(iplication), operator **prod**(uct), operator **if**/**then**/**else**, and operator **let**/**in**. Operator **prod** binds its natural number identifier to every value in its range and forms the product of its body for each of those bindings (cf. looping constructs in imperative languages); see also the denotational semantics below. The FOCAML syntax has no explicit operator for subtraction. Instead, when interpreting a

```
SyncK[k](in;out) = {
  if (k == 1) {
    Sync(in;out)
  } else {
    Sync(in;P[1])
      mult { prod i:1..k-2 { Sync(P[i];P[i+1]) } }
        mult Sync(P[k-1];out)
} }

FifoK[k](in;out) = {
  if (k == 1) {
    Fifo(in;out)
  } else {
    Fifo(in;P[1])
      mult { prod i:1..k-2 { Fifo(P[i];P[i+1]) } }
        mult Fifo(P[k-1];out)
} }

Merger(in[];out) = {
  let k = #in {
    if (k == 1) {
      Sync(in[1];out)
    } else if (k == 2) {
      Merger2(in[1],in[2];out)
    } else {
      Merger2(in[1],in[2];P[2])
        mult { prod i:3..k-1 { Merger2(P[i-1],in[i];P[i]) } }
          mult Merger2(P[k-1],in[k];out)
} } }

Router(in;out[]) = {
  let k = #out {
    if (k == 1) {
      Sync(in;out[1])
    } else if (k == 2) {
      Router2(in;out[1],out[2])
    } else {
      Router2(P[2];out[1],out[2])
        mult { prod i:3..k-1 { Router2(P[i];P[i-1],out[i]) } }
          mult Router2(in;P[k-1],out[k])
} } }

LateAsyncMerger(in[];out) = { Merger(in[1..#in];P) mult Fifo(P;out) }

EarlyAsyncMerger(in[];out) = {
  let k = #in {
    { prod i:1..k { Fifo(in[i];P[i]) } } mult Merger(P[1..k];out)
} }
```

Figure 3.12: FOCAML definitions for families SyncK, FifoK, Merger, LateAsyncMerger, and EarlyAsyncMerger

```
1  main = {
2      LateAsyncMerger2(A,B;C)
3    } among {
4      Producer(A) and Producer(B) and Consumer(C)
5    }
```

Figure 3.13: Producers/consumer program for LateAsyncMerger2 in FOCAML (cf. Figures 1.4 and 1.10)

family definition, a FOCAML interpreter automatically subtracts all ports that occur in the body of that definition but not in its signature (cf. local variables). Henceforth, I call such ports *local ports*. As a notational convention, I write identifiers for ports/arrays in lowercase, while I capitalize actual values for ports/arrays. Figures 3.3, 3.11, and 3.12 show example bodies (cf. Figures 3.2, 3.7, and 3.10). Note that Figure 3.12 also shows k-parametric versions of LateAsyncMerger2 and EarlyAsyncMerger2 in Figure 3.3.

- A main definition consists of an optional list of program arguments (at run-time passed via the command line) and a main body. The main body consists of a list of instantiated family signatures and, separated by keyword **among**, an optional list of instantiated *foreign* signatures. Every instantiated family signature in the former list refers to a protocol subprogram in FOCAML (i.e., a family definition); if present, every foreign signature in the latter list refers to a worker subprogram in a complementary GPL (e.g., a public static method in Java or a function in C). By sharing ports between family signatures and foreign signatures, the main body states which links exist at run-time between protocols and workers. Mainly for testing purposes, software engineers may omit the list of foreign signatures; doing so does not yield a comprehensive program, but my FOCAML interpreter (briefly described later in this chapter) and compiler (in detail discussed in Chapters 4–8) will still process the list of instantiated family signatures and generate code for them.

  Figure 3.13 shows an example of a main subprogram for the producers/consumer example in Chapter 1. I can nearly effortlessly change the protocol in this program just by replacing `LateAsyncMerger2` with, for instance, `EarlyAsyncMerger2` in the main body. This shows that FOCAML enables software engineers to easily change implementations of protocol specifications without affecting computation code (cf. Parnas' advantages of modularization in Chapter 1).

FOCAML has more features, including a basic macro system and constructs for supplementing the core set with new families of constraint automata. I do not discuss these practically important but theoretically insignificant features here.

Before giving a precise definition of the denotational semantics of FOCAML, I first explain this semantics more informally and by example. Recall that

a FOCAML program represents a multiplication expression of constraint automata. With that in mind, I define the denotational semantics of instantiated family signatures (in the main body or in the body of a family definition) inductively over the set of all constraint automata. For the base case, every instantiated signature of ⌈a core set family defined in Figure 3.4⌉ denotes the corresponding member of that family. Inductively, then, every instantiated signature of ⌈a family defined in the program text⌉ denotes the multiplication of the denotations of the instantiated signatures in its body, minus local ports. For instance, `LateAsyncMerger2(A,B;C)` has the following denotation (including automatic subtraction of the ports denoted by `P1`, `P2`, and `P3`):

$$
\begin{aligned}
&[\![\texttt{LateAsyncMerger2(A,B;C)}]\!] \\
&= \left(\begin{array}{l} [\![\texttt{Sync(A;P1)}]\!] \\ \otimes\, [\![\texttt{Sync(B;P2)}]\!] \\ \otimes\, [\![\texttt{Merger2(P1,P2;P3)}]\!] \\ \otimes\, [\![\texttt{Fifo(P3;C)}]\!] \end{array} \right) \ominus [\![\texttt{P1}]\!] \ominus [\![\texttt{P2}]\!] \ominus [\![\texttt{P3}]\!] \\
&= \left(\begin{array}{l} \mathsf{Sync(A;P1)} \\ \otimes\, \mathsf{Sync(B;P2)} \\ \otimes\, \mathsf{Merger2(P1,P2;P3)} \\ \otimes\, \mathsf{Fifo(P3;C)} \end{array} \right) \ominus \mathsf{P1} \ominus \mathsf{P2} \ominus \mathsf{P3}
\end{aligned}
$$

For the last step in this derivation to hold true, trivially let `A`, `B`, `C`, `P1`, `P2`, and `P3` denote A, B, C, P1, P2, and P3. For the actual computation of the final multiplication expression, see Figures 2.5 and 2.6. As another example, `LateAsyncMerger(A[1..2];C)` has the following denotation (including automatic subtraction of the ports denoted by `P[1]`, `P[2]`, and `P`):

$$
\begin{aligned}
&[\![\texttt{LateAsyncMerger(A[1..2];C)}]\!] \\
&= ([\![\texttt{Merger(A[1..2];P)}]\!] \otimes [\![\texttt{Fifo(P;B)}]\!]) \ominus [\![\texttt{P}]\!] \\
&= \left(\left(\begin{array}{l} \left(\begin{array}{l} [\![\texttt{Sync(A[1];P[1])}]\!] \\ \otimes\, [\![\texttt{Merger2(A[2],P[1];P[2])}]\!] \end{array} \right) \\ \otimes\, [\![\texttt{Sync(P[2];P)}]\!] \end{array} \right) \ominus [\![\texttt{P[1]}]\!] \ominus [\![\texttt{P[2]}]\!] \right) \otimes [\![\texttt{Fifo(P;B)}]\!] \right) \ominus [\![\texttt{P}]\!] \\
&\simeq \left(\begin{array}{l} [\![\texttt{Sync(A[1];P[1])}]\!] \\ \otimes\, [\![\texttt{Merger2(A[2],P[1];P[2])}]\!] \\ \otimes\, [\![\texttt{Sync(P[2];P)}]\!] \\ \otimes\, [\![\texttt{Fifo(P;B)}]\!] \end{array} \right) \ominus [\![\texttt{P[1]}]\!] \ominus [\![\texttt{P[2]}]\!] \ominus [\![\texttt{P}]\!]
\end{aligned}
$$

The last step in this derivation holds true, because the denotations of `P[1]` and `P[2]` do not occur in the constraint automaton denoted by `Fifo(P;B)`. In those cases, subtraction can move outward by Theorem 5.

Constraint automata have enough expressive power for modeling computation. Sirjani et al., for instance, used the "original constraint automata" by Baier et al.—a subset of the constraint automata in this thesis—to model actors [SJBA06]. Technically, the extent of this expressiveness depends on the

$\texttt{I} \in \texttt{Identifier}$	
$\texttt{B} \in \texttt{Boolean}$	$\texttt{BE} \in \texttt{BooleanExpression}$
$\texttt{N} \in \texttt{Natural}$	$\texttt{NE} \in \texttt{NaturalExpression}$
$\texttt{E} \in \texttt{Extralogical}$	$\texttt{EE} \in \texttt{ExtralogicalExpression}$
$\texttt{P} \in \texttt{Port}$	$\texttt{PE} \in \texttt{PortExpression}$
$\texttt{Ar} \in \texttt{Array}$	$\texttt{ArE} \in \texttt{ArrayExpression}$
	$\texttt{?E} \in \texttt{N/E/P/ArExpression}$
$\texttt{FD} \in \texttt{FamilyDefinition}$	$\texttt{AE} \in \texttt{AutomatonExpression}$
$\texttt{MD} \in \texttt{MainDefinition}$	
$\texttt{R} \in \texttt{Program}$	

Figure 3.14: Abstract syntax domains of FOCAML

$$
\begin{aligned}
\texttt{BE} &::= \texttt{B} \mid \texttt{NE}_1 \texttt{ == } \texttt{NE}_2 \mid \texttt{!BE} \mid \texttt{BE}_1 \texttt{ \&\& } \texttt{BE}_2 \mid \texttt{BE}_1 \texttt{ || } \texttt{BE}_2 \\
\texttt{NE} &::= \texttt{N} \mid \texttt{I} \mid \texttt{NE}_1 \texttt{ + } \texttt{NE}_2 \mid \texttt{NE}_1 \texttt{ - } \texttt{NE}_2 \mid \texttt{NE}_1 \texttt{ * } \texttt{NE}_2 \mid \texttt{NE}_1 \texttt{ / } \texttt{NE}_2 \mid \texttt{NE}_1 \texttt{ \% } \texttt{NE}_2 \mid \texttt{\#I} \\
\texttt{EE} &::= \texttt{E} \mid \texttt{I} \\
\texttt{PE} &::= \texttt{P} \mid \texttt{Ar[NE]} \mid \texttt{I} \mid \texttt{I[NE]} \\
\texttt{ArE} &::= \texttt{[PE}_1\texttt{,} \ldots \texttt{,PE}_k\texttt{]} \mid \texttt{Ar[NE}_1\texttt{..NE}_2\texttt{]} \mid \texttt{I} \mid \texttt{I[NE}_1\texttt{..NE}_2\texttt{]} \\
\texttt{?E} &::= \texttt{NE} \mid \texttt{EE} \mid \texttt{PE} \mid \texttt{ArE} \\
\texttt{AE} &::= \texttt{I ?E}_1 \cdots \texttt{?E}_k \\
&\quad \mid \texttt{AE}_1 \textbf{ mult } \texttt{AE}_2 \\
&\quad \mid \textbf{prod } \texttt{I:NE}_1\texttt{..NE}_2 \texttt{ AE} \\
&\quad \mid \textbf{if } \texttt{BE } \textbf{then } \texttt{AE}_1 \textbf{ else } \texttt{AE}_2 \\
&\quad \mid \textbf{let } \texttt{I = NE AE} \\
\texttt{FD} &::= \texttt{I I}_1 \cdots \texttt{I}_k \texttt{ = AE} \\
\texttt{MD} &::= \textbf{main} \texttt{ = AE} \\
\texttt{G} &::= \texttt{FD G} \mid \texttt{G FD} \mid \texttt{MD}
\end{aligned}
$$

Figure 3.15: Abstract syntax of FOCAML

set of all data $\mathbb{D}$. For instance, if $\mathbb{D}$ contains data structures for unbounded tapes, constraint automata can simulate Turing machines. With such a $\mathbb{D}$, I can model the behavior of worker subprograms referenced by instantiated foreign signatures in a FOCAML program as constraint automata. The denotational semantics of the main definition of that program then consists of a multiplication of the constraint automata for its instantiated family and foreign signatures. I discuss the behavior of main definitions in a more operationally in Chapter 4.

Having presented the denotational semantics of FOCAML informally, I now make it precise. Figure 3.14 shows the domains for FOCAML's abstract syn-

tax; Figure 3.15 shows its abstract syntax. I use this abstract syntax only in FOCAML's denotational semantics, below, so it captures only the *essence* of what FOCAML programs represent: (multiplications of) constraint automata. With this goal in mind, Figure 3.15 applies two notable abstractions to the concrete syntax [**Jon**16]): (i) signatures consist of one unbracketed/unordered list of implicitly typed identifiers for natural numbers, extralogicals, ports, and arrays and (ii) main definitions consist of an expression of constraint automata, thereby abstracting away the concrete distinction between protocols and workers, each of which semantically just denotes a constraint automaton.

Figure 3.16 shows the domains for FOCAML's denotational semantics; Figures 3.17–3.20 show its denotational semantics, annotated with comments to clarify their definitions. Some additional general remarks:

- Recall that family definitions have no recursion (for simplicity and because no theoretical need for it seems to exist), and note that this denotational semantics evaluates family definitions in an eager fashion (to detect erroneous definitions early and because FOCAML has no infinite expansion).

- In Figure 3.19, let prim denote a function that maps identifiers to families of primitive constraint automata while generating fresh memory cells by need; under the core set in this thesis, prim maps identifiers to the families of constraint automata in Figure 3.4. For instance, under Figure 3.4, prim contains (among others):

  $$\mathsf{I}[\![\texttt{Sync}]\!] \mapsto \lambda p_1.\lambda p_2.\mathsf{Sync}(p_1;p_2)$$

  and:

  $$\mathsf{I}[\![\texttt{Fifo}]\!] \mapsto \lambda p_1.\lambda p_2.\mathsf{Fifo}\{;\mathsf{fresh}\}(p_1;p_2)$$

  where fresh denotes a fresh memory cell. Technically, fresh abstracts away a straightforward bookkeeping mechanism in the denotational semantics that ensures uniqueness of memory cells in different constraint automata. For families with more than one memory cell parameter, I "invoke" fresh separately for each of those parameters (e.g., $\mathsf{Fifo2}\{;\mathsf{fresh},\mathsf{fresh}\}(p_1;p_2)$), although in such cases, technically, fresh needs an extra parameter to distinguish the second "invocation" from the first one.

- I stipulate that whenever one of the subphrases of a phrase has no denotation, that phrase itself has no denotation either. Similarly, I stipulate that whenever the formal and actual parameters of an instantiated signature do not match, that instantiated signature has no denotation. A type checker may detect such errors already before actually evaluating the denotational semantics of a program. Finally, every identifier for an input/output port in a signature of a family definition should occur exactly once in the body of that family definition, while every local port in a body should occur at most twice in that body: at most once as an input port and at most once as an output port. Otherwise, I stipulate that this

- **Identifiers**, $\mathbb{I}$
- **Booleans**, $\mathbb{B} = \{\mathrm{false}, \mathrm{true}\}$
  - $\doteq : \mathbb{N} \times \mathbb{N} \to \mathbb{B}$ (equality on naturals)
  - $\neg : \mathbb{B} \to \mathbb{B}$ (negation)
  - $\wedge, \vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (conjunction, disjunction)
- **Naturals**, $\mathbb{N} = \{0, 1, 2, \ldots\}$
  - $+, \times : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ (addition, multiplication)
  - $-, \div, \mathrm{mod} : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ (subtraction, division, modulo)
- **Extralogicals**, $\mathbb{E}\mathrm{xtr} = \mathbb{D} \cup \mathbb{F} \cup \mathbb{R}$
- **Ports**, $\mathbb{P}$
- **Arrays**, $\mathbb{P}[] = \mathbb{N} \rightharpoonup \mathbb{P}$
  - $\emptyset : \mathbb{P}[]$ (new)
  - $\cdot\{\cdot \mapsto \cdot\} : \mathbb{P}[] \times \mathbb{N} \times \mathbb{P} \to \mathbb{P}[]$ (mutator)
  - $\cdot[\cdot] : \mathbb{P}[] \times \mathbb{N} \rightharpoonup \mathbb{P}$ (accessor)
  - $|\cdot| : \mathbb{P}[] \to \mathbb{N}$ (length)
- **Constraint automata**, $\mathbb{A}\mathrm{UTOM}$
  - $\otimes : \mathbb{A}\mathrm{UTOM} \times \mathbb{A}\mathrm{UTOM} \rightharpoonup \mathbb{A}\mathrm{UTOM}$ (multiplication)
  - $\ominus : \mathbb{A}\mathrm{UTOM} \times \mathbb{P} \to \mathbb{A}\mathrm{UTOM}$ (subtraction)
- **Environment values**, $\mathbb{E}\mathrm{nv}\mathbb{V}\mathrm{al} = \mathbb{N} \cup \mathbb{E}\mathrm{xtr} \cup \mathbb{P} \cup \mathbb{P}[] \cup \mathbb{F}\mathrm{AM}$
- **Environments**, $\mathbb{E}\mathrm{nv} = \mathbb{I} \rightharpoonup \mathbb{E}\mathrm{nv}\mathbb{V}\mathrm{al}$
  - $\emptyset : \mathbb{E}\mathrm{nv}$ (new)
  - $\cdot\{\cdot \mapsto \cdot\} : \mathbb{E}\mathrm{nv} \times \mathbb{I} \times \mathbb{E}\mathrm{nv}\mathbb{V}\mathrm{al} \to \mathbb{E}\mathrm{nv}$ (mutator)
  - $(\cdot\,\cdot) : \mathbb{E}\mathrm{nv} \times \mathbb{I} \rightharpoonup \mathbb{E}\mathrm{nv}\mathbb{V}\mathrm{al}$ (accessor)

Figure 3.16: Denotational semantics domains

body has no denotation. Again, a type checker may detect violation of this requirement. I leave developing the theory of such a type checker for future work.

- I stipulate that structural disambiguation of (similarly named) local ports in different family definitions has occurred already before evaluating the denotational semantics of a program.

- $\mathsf{I}[\![\cdot]\!] : \texttt{Identifier} \to \mathbb{I}$ is a bijection.
- $\mathsf{B}[\![\cdot]\!] : \texttt{Boolean} \to \mathbb{B}$ is a bijection.
- $\mathsf{BE}[\![\cdot]\!] : \texttt{BooleanExpression} \to \mathbb{E}\mathrm{nv} \rightharpoonup \mathbb{B}$
  $\mathsf{BE}[\![\texttt{B}]\!] = \lambda e.\mathsf{B}[\![\texttt{B}]\!]$
  $\mathsf{BE}[\![\texttt{I}]\!] = \lambda e.(e\ \mathsf{I}[\![\texttt{I}]\!])\ \textbf{if}\ (e\ \mathsf{I}[\![\texttt{I}]\!]) \in \mathbb{B}$
  $\mathsf{BE}[\![\texttt{NE}_1\ \texttt{==}\ \texttt{NE}_2]\!] = \lambda e.((\mathsf{NE}[\![\texttt{NE}_1]\!]\ e) = (\mathsf{NE}[\![\texttt{NE}_2]\!]\ e))$
  $\mathsf{BE}[\![\texttt{!BE}]\!] = \lambda e.\neg(\mathsf{BE}[\![\texttt{BE}]\!]\ e)$
  $\mathsf{BE}[\![\texttt{BE}_1\ \texttt{\&\&}\ \texttt{BE}_2]\!] = \lambda e.((\mathsf{BE}[\![\texttt{BE}_1]\!]\ e) \wedge (\mathsf{BE}[\![\texttt{BE}_2]\!]\ e))$
  $\mathsf{BE}[\![\texttt{BE}_1\ \texttt{||}\ \texttt{BE}_2]\!] = \lambda e.((\mathsf{BE}[\![\texttt{BE}_1]\!]\ e) \vee (\mathsf{BE}[\![\texttt{BE}_2]\!]\ e))$
- $\mathsf{N}[\![\cdot]\!] : \texttt{Natural} \to \mathbb{N}$ is a bijection.
- $\mathsf{NE}[\![\cdot]\!] : \texttt{NaturalExpression} \to \mathbb{E}\mathrm{nv} \rightharpoonup \mathbb{N}$
  $\mathsf{NE}[\![\texttt{N}]\!] = \lambda e.\mathsf{N}[\![\texttt{N}]\!]$
  $\mathsf{NE}[\![\texttt{I}]\!] = \lambda e.(e\ \mathsf{I}[\![\texttt{I}]\!])\ \textbf{if}\ (e\ \mathsf{I}[\![\texttt{I}]\!]) \in \mathbb{N}$
  $\mathsf{NE}[\![\texttt{NE}_1\ \texttt{+}\ \texttt{NE}_2]\!] = \lambda e.((\mathsf{NE}[\![\texttt{NE}_1]\!]\ e) + (\mathsf{NE}[\![\texttt{NE}_2]\!]\ e))$
  $\mathsf{NE}[\![\texttt{NE}_1\ \texttt{*}\ \texttt{NE}_2]\!] = \lambda e.((\mathsf{NE}[\![\texttt{NE}_1]\!]\ e) \times (\mathsf{NE}[\![\texttt{NE}_2]\!]\ e))$
  $\mathsf{NE}[\![\texttt{NE}_1\ \texttt{-}\ \texttt{NE}_2]\!] = \lambda e.((\mathsf{NE}[\![\texttt{NE}_1]\!]\ e) - (\mathsf{NE}[\![\texttt{NE}_2]\!]\ e))$
  $\mathsf{NE}[\![\texttt{NE}_1\ \texttt{/}\ \texttt{NE}_2]\!] = \lambda e.((\mathsf{NE}[\![\texttt{NE}_1]\!]\ e) \div (\mathsf{NE}[\![\texttt{NE}_2]\!]\ e))$
  $\mathsf{NE}[\![\texttt{NE}_1\ \texttt{\%}\ \texttt{NE}_2]\!] = \lambda e.((\mathsf{NE}[\![\texttt{NE}_1]\!]\ e) \bmod (\mathsf{NE}[\![\texttt{NE}_2]\!]\ e))$
  $\mathsf{NE}[\![\texttt{\#I}]\!] = \lambda e.|e\ \mathsf{I}[\![\texttt{I}]\!]|\ \textbf{if}\ (e\ \mathsf{I}[\![\texttt{I}]\!]) \in \mathbb{P}[]$
- $\mathsf{E}[\![\cdot]\!] : \texttt{Extralogical} \to \mathbb{E}\mathrm{xtr}$ is a bijection.
- $\mathsf{EE}[\![\cdot]\!] : \texttt{ExtralogicalExpression} \to \mathbb{E}\mathrm{nv} \rightharpoonup \mathbb{E}\mathrm{xtr}$
  $\mathsf{EE}[\![\texttt{E}]\!] = \lambda e.\mathsf{E}[\![\texttt{E}]\!]$
  $\mathsf{EE}[\![\texttt{I}]\!] = \lambda e.(e\ \mathsf{I}[\![\texttt{I}]\!])\ \textbf{if}\ (e\ \mathsf{I}[\![\texttt{I}]\!]) \in \mathbb{E}\mathrm{xtr}$

Figure 3.17: Denotational semantics (I)

FOCAML comprises a declarative, textual syntax for multiplication expressions of constraint automata. From a different point of view, one may also present FOCAML as a textual version of Reo; Baier et al. and Klüppelholz developed an alternative, imperative textual syntax of Reo [BBKK09a, Klü12], called *Reo Scripting Language* (RSL). RSL originated from research on *Vereofy* [BBK+10, BBKK09a, BBKK09b, BKK11, KB09, KKSB11], a model checker for Reo based on constraint automata. In RSL, software engineers write exactly *how* to construct a particular Reo circuit rather than expressing *what* that circuit constitutes, as in FOCAML. This imperative style of programming in RSL makes RSL more verbose than FOCAML and comprises the main fundamental difference between these two languages. As an example of this difference, Figure 3.21 shows a member of the $\mathsf{Sequencer}_4$ family; Figure 3.22 shows a circuit for that same member; Figure 3.23 shows a FOCAML definition for the entire Sequencer family and a main definition for the same member as in the previous

- $\mathsf{P}[\![\cdot]\!] : \texttt{Port} \to \mathbb{P}$ is an injection.
  P maps every P (the name of a port in a program text) to a port in $\mathbb{P}$. I explain the injectivity of P below, when I discuss Ar.

- $\mathsf{PE}[\![\cdot]\!] : \texttt{PortExpression} \to \mathbb{E}\mathrm{nv} \rightharpoonup \mathbb{P}$
  $\mathsf{PE}[\![\texttt{P}]\!] = \lambda e.\mathsf{P}[\![\texttt{P}]\!]$
  $\mathsf{PE}[\![\texttt{Ar [NE]}]\!] = \lambda e.\mathsf{Ar}[\![\texttt{Ar}]\!][\mathsf{NE}[\![\texttt{NE}]\!]\ e]$
  $\mathsf{PE}[\![\texttt{I}]\!] = \lambda e.(e\ \mathsf{I}[\![\texttt{I}]\!])$ **if** $(e\ \mathsf{I}[\![\texttt{I}]\!]) \in \mathbb{P}$
  $\mathsf{PE}[\![\texttt{I [NE]}]\!] = \lambda e.(e\ \mathsf{I}[\![\texttt{I}]\!])[\mathsf{NE}[\![\texttt{NE}]\!]\ e]$ **if** $(\mathsf{NE}[\![\texttt{NE}]\!]\ e) \in \mathrm{Dom}(e\ \mathsf{I}[\![\texttt{I}]\!])$

  PE either maps a PE and a curried environment e to a port, or PE has no denotation under e. PE uses e [to get a port or an array for an I] and/or [to get an array index denoted by an NE]. Note that the side condition for I[I] in the fourth equation implies $(e\ \mathsf{I}[\![\texttt{I}]\!]) \in \mathbb{P}[]$, analogous to the side conditions for I in the third equation.

- $\mathsf{Ar}[\![\cdot]\!][\cdot] : (\texttt{Array} \times \mathbb{N}) \to \mathbb{P} \setminus \mathrm{Img}(\mathsf{P})$ is an injection.
  Ar maps every pair of an Ar (the name of an array of ports in a program text) and a natural number (an index in that array) to a port in $\mathbb{P} \setminus \mathrm{Img}(\mathsf{P})$. The exclusion of the image of P from the codomain of Ar ensures, together with injectivity (and an implicit assumption that $\mathbb{P} \setminus \mathrm{Img}(\mathsf{P})$ has countably many elements), that P and Ar map their arguments to unique ports.

- $\mathsf{ArE}[\![\cdot]\!] : \texttt{ArrayExpression} \to \mathbb{E}\mathrm{nv} \rightharpoonup \mathbb{P}[]$
  $\mathsf{ArE}[\![\texttt{[PE}_1\texttt{, ... ,PE}_k\texttt{]}]\!] = \lambda e.\emptyset\{1 \mapsto (\mathsf{PE}[\![\texttt{PE}_1]\!]\ e)\}\cdots\{k \mapsto (\mathsf{PE}[\![\texttt{PE}_k]\!]\ e)\}$
  $\mathsf{ArE}[\![\texttt{Ar [NE}_1\texttt{..NE}_2\texttt{]}]\!] = \lambda e.\emptyset\{1 \mapsto (\mathsf{PE}[\![\texttt{Ar [NE}_1\texttt{]}]\!]\ e)\}$
  $\cdots$
  $\{\mathsf{NE}[\![\texttt{\#Ar [NE}_1\texttt{..NE}_2\texttt{]}]\!] \mapsto (\mathsf{PE}[\![\texttt{Ar [NE}_2\texttt{]}]\!]\ e)\}$
  $\mathsf{ArE}[\![\texttt{I}]\!] = \lambda e.(e\ \mathsf{I}[\![\texttt{I}]\!])$ **if** $(e\ \mathsf{I}[\![\texttt{I}]\!]) \in \mathbb{P}[]$
  $\mathsf{ArE}[\![\texttt{I [NE}_1\texttt{..NE}_2\texttt{]}]\!] =$
  $\lambda e.\emptyset\{1 \mapsto (\mathsf{PE}[\![\texttt{I [NE}_1\texttt{]}]\!]\ e)\}\cdots\{\mathsf{NE}[\![\texttt{\#Ar [NE}_1\texttt{..NE}_2\texttt{]}]\!] \mapsto (\mathsf{PE}[\![\texttt{I [NE}_2\texttt{]}]\!]\ e)\}$

  ArE either maps an ArE and a curried environment e to an array, or ArE has no denotation under e. If one of ArE's subphrases has no denotation, also ArE itself has no denotation. Array indices start at 1.

- $\mathsf{?E}[\![\cdot]\!] : (\texttt{N/E/P/ArExpression} \to \mathbb{E}\mathrm{nv} \rightharpoonup \mathbb{N} \cup \mathbb{E}\mathrm{xtr} \cup \mathbb{P} \cup \mathbb{P}[]$
  $\mathsf{?E}[\![\texttt{NE}]\!] = \lambda e.\mathsf{NE}[\![\texttt{NE}]\!]$
  $\mathsf{?E}[\![\texttt{EE}]\!] = \lambda e.\mathsf{EE}[\![\texttt{EE}]\!]$
  $\mathsf{?E}[\![\texttt{PE}]\!] = \lambda e.\mathsf{PE}[\![\texttt{PE}]\!]$
  $\mathsf{?E}[\![\texttt{ArE}]\!] = \lambda e.\mathsf{ArE}[\![\texttt{ArE}]\!]$

  ?E consists of the union of NE, EE, PE and ArE.

Figure 3.18: Denotational semantics (II)

- $\mathsf{AE}[\![\cdot]\!] : \texttt{AutomatonExpression} \to \mathbb{E}\mathrm{nv} \rightharpoonup \mathbb{A}\mathrm{UTOM}$
  $\mathsf{AE}[\![\texttt{I ?E}_1 \cdots \texttt{?E}_k]\!] = \lambda e.((\mathsf{prim} \cup e)\ \mathsf{I}[\![\texttt{I}]\!]\ (\mathsf{?E}[\![\texttt{?E}_1]\!]\ e) \cdots (\mathsf{?E}[\![\texttt{?E}_k]\!]\ e))$
  $\mathsf{AE}[\![\texttt{AE}_1\ \textbf{mult}\ \texttt{AE}_2]\!] = \lambda e.((\mathsf{AE}[\![\texttt{AE}_1]\!]\ e) \otimes (\mathsf{AE}[\![\texttt{AE}_2]\!]\ e))$
  $\mathsf{AE}[\![\textbf{prod}\ \texttt{I:NE}_1\texttt{..NE}_2\ \texttt{AE}]\!] = \lambda e.(\boldsymbol{a}_1 \otimes \cdots \otimes \boldsymbol{a}_z)\ \textbf{if}\ z > 0$
  $$\textbf{for}\ z = 1 + (\mathsf{NE}[\![\texttt{NE}_2]\!]\ e) - (\mathsf{NE}[\![\texttt{NE}_1]\!]\ e)$$
  $$\textbf{and}\ \boldsymbol{a}_1 = \mathsf{AE}[\![\texttt{AE}]\!]\ e\{\mathsf{I}[\![\texttt{I}]\!] \mapsto \mathsf{NE}[\![\texttt{NE}_1]\!]\ e\}$$
  $$\textbf{and}\ \cdots$$
  $$\textbf{and}\ \boldsymbol{a}_z = \mathsf{AE}[\![\texttt{AE}]\!]\ e\{\mathsf{I}[\![\texttt{I}]\!] \mapsto \mathsf{NE}[\![\texttt{NE}_2]\!]\ e\}$$
  $\mathsf{AE}[\![\textbf{if}\ \texttt{BE}\ \textbf{then}\ \texttt{AE}_1\ \textbf{else}\ \texttt{AE}_2]\!] = \lambda e. \begin{cases} (\mathsf{AE}[\![\texttt{AE}_1]\!]\ e)\ \textbf{if}\ (\mathsf{BE}[\![\texttt{BE}]\!]\ e) = \text{true} \\ (\mathsf{AE}[\![\texttt{AE}_2]\!]\ e)\ \textbf{if}\ (\mathsf{BE}[\![\texttt{BE}]\!]\ e) = \text{false} \end{cases}$
  $\mathsf{AE}[\![\textbf{let}\ \texttt{I = NE}\ \texttt{AE}]\!] = \lambda e.(\mathsf{AE}[\![\texttt{AE}]\!]\ e\{\mathsf{I}[\![\texttt{I}]\!] \mapsto (\mathsf{NE}[\![\texttt{NE}]\!]\ e)\})$

  AE either maps an AE and a curried environment e to a constraint automaton, or AE has no denotation under e. If one of AE's subphrases has no denotation, also AE itself has no denotation. Similarly, if the formal and actual parameters in AE do not match (i.e., the first equation), AE has no denotation.

- $\mathsf{FD}[\![\cdot]\!] : \texttt{FamilyDefinition} \to \mathbb{E}\mathrm{nv} \rightharpoonup \mathbb{E}\mathrm{nv}$
  $\mathsf{FD}[\![\texttt{I I}_1 \cdots \texttt{I}_k\ \texttt{= AE}]\!] =$
  $$\lambda e.e \left\{ \begin{array}{l} \mathsf{I}[\![\texttt{I}]\!] \mapsto \lambda\sharp_1.\cdots\lambda\sharp_k.((\mathsf{AE}[\![\texttt{AE}]\!]\ e') \ominus (\mathbb{P} \setminus (P \cup P[])) \\ \qquad \textbf{for}\ e' = e\{\mathsf{I}[\![\texttt{I}_1]\!] \mapsto \sharp_1\} \cdots \{\mathsf{I}[\![\texttt{I}_k]\!] \mapsto \sharp_k\} \\ \qquad\quad \textbf{and}\ P = \{\sharp \mid \sharp \in \mathrm{Img}(e') \cap \mathbb{P}\} \\ \qquad\quad \textbf{and}\ P[] = \bigcup\{\mathrm{Img}(\sharp) \mid \sharp \in \mathrm{Img}(e') \cap \mathbb{P}[]\} \end{array} \right\}$$

  FD either maps an FD and a curried environment e to a new, extended environment, or FD has no denotation under e. If one of FD's subphrases has no denotation, also FD itself has no denotation. Otherwise, FD adds a new mapping to e, from identifier $\mathsf{I}[\![\texttt{I}]\!]$ to a function that consumes a list of natural numbers, extralogicals, ports, and port arrays as input, all ranged over by $\sharp$, and produces a constraint automaton as output. This function, then, essentially defines a family of constraint automata. Each of the formal parameters—the identifiers denoted by $\texttt{I}_1$ up to $\texttt{I}_k$—become bound to an actual parameter $\sharp_i$ in environment e'. P and $P[]$ contain the ports in $\sharp_1, \ldots, \sharp_k$: the former contains individual ports, while the latter contains ports in arrays. Abusing notation (technically, $\ominus$ takes individual ports as input instead of sets), FD subtracts all nonparameter ports outside P and $P[]$ from the denotation of AE.

Figure 3.19: Denotational semantics (III)

- $\mathsf{MD}[\![\cdot]\!] : \texttt{MainDefinition} \to \mathbb{E}\mathrm{nv} \rightharpoonup \mathbb{A}\mathrm{UTOM}$
  $\mathsf{MD}[\![\textbf{main} = \texttt{AE}]\!] = \lambda e.(\mathsf{AE}[\![\texttt{AE}]\!]\ e)$
- $\mathsf{G}[\![\cdot]\!] : \texttt{Program} \to \mathbb{E}\mathrm{nv} \rightharpoonup \mathbb{A}\mathrm{UTOM}$
  $\mathsf{G}[\![\texttt{FD G}]\!] = \lambda e.(\mathsf{G}[\![\texttt{G}]\!]\ (\mathsf{FD}[\![\texttt{FD}]\!]\ e))$
  $\mathsf{G}[\![\texttt{G FD}]\!] = \lambda e.(\mathsf{G}[\![\texttt{G}]\!]\ (\mathsf{FD}[\![\texttt{FD}]\!]\ e))$
  $\mathsf{G}[\![\texttt{MD}]\!] = \lambda e.(\mathsf{MD}[\![\texttt{MD}]\!]\ e)$

Figure 3.20: Denotational semantics (IV)

figures; Figure 3.24 shows an RSL definition, taken from the Vereofy user manual [BKK10]. The instantiated family signature `Sequencer([A,B,C,D];)` has a denotation behaviorally equivalent to the constraint automaton in Figure 3.21 but not structurally equal. In particular, I simplified the data constraints in the denotation of `Sequencer([A,B,C,D];)`, for the sake of clarity and presentation, to obtain the constraint automaton in Figure 3.21.

Members of Sequencer impose an order in which puts of workers can complete: first a put on A completes, then one on B, then one on C, and finally one on D. The FOCAML definition in Figure 3.23 expresses this protocol in a declarative style, as an essentially mathematical multiplication expression. The RSL definition in Figure 3.24, in contrast, expresses this protocol in an imperative style (i.e., first create a full Fifo, then create a SyncDrain, then join the sink end of the full Fifo with the second source end of the SyncDrain on a node, etc.), as witnessed by the use of sequential composition (semicolon) and the **for** keyword for repetition. Other differences between FOCAML and RSL include use of signatures and linguistic emphasis on constraint automata (in FOCAML) or Reo (in RSL).

3.2 Practice

(I have not yet submitted the material in this section for publication.)

Editor

I developed an editor/parser/interpreter for FOCAML as a plugin for Eclipse 4.x. The editor has basic features such as syntax highlighting and error reporting. To provide this functionality, the editor automatically invokes a FOCAML parser (written with ANTLR) and a FOCAML interpreter every time the user makes changes to a FOCAML program. The FOCAML interpreter implements the denotational semantics in the previous subsection by generating a list of primitive constraint automata for every instantiated family signature in the main definition of a FOCAML program. Such a list represents a multiplication expression over those primitives, similar to the last line in the example deriva-

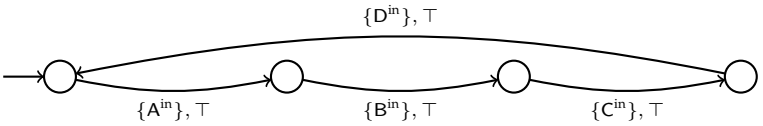


Figure 3.21: Constraint automaton for the $\mathsf{Sequencer}_4$ protocol. The first worker has access to port A, the second to port B, the third to port C, and the fourth to port D.

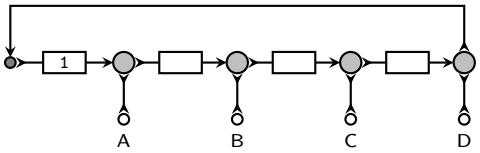


Figure 3.22: Circuit for a member of subfamily $\mathsf{Sequencer}_4$

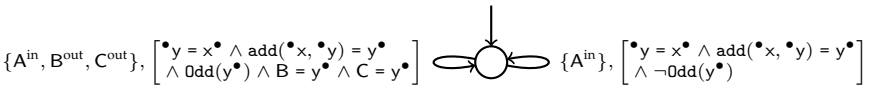
```
1  Sequencer(in[];) = {
2    FifoFull<'1'>(P1[1];P2[1])
3    mult { prod i:2..#in { Fifo(P1[i];P2[i]) } }
4      mult { prod i:1..#in { Replicator2(P2[i];P1[i+1],P3[i]) } }
5        mult { prod i:1..#in { SyncDrain(in[i],P3[i];) } }
6      mult Sync(P1[#in+1];P1[1])
7  }

8  main = { Sequencer([A,B,C,D];) }
```

Figure 3.23: FOCAML definition for family Sequencer

```
1  CIRCUIT SEQUENCER<k> {
2    F[0] = new FIFO1_FULL<1>(A[0];B[0]);
3    SD[0] = new SYNC_DRAIN(C[0],D[0]);
4    Node[0] = join(B[0],D[0]);
5    for (i = 1; i < k; i = i + 1) {
6      F[i] = new FIFO1(A[i];B[i]);
7      SD[i] = new SYNC_DRAIN(C[i],D[i]);
8      Node[i] = join(B[i],D[i]);
9      Node[i-1] = join(Node[i-1],A[i]);
10   }
11   Node[k-1] = join(Node[k-1],A[0]);
12   for (i = 0; i < k; i = i + 1) {
13     source[i] = C[i];
14 } }
```

Figure 3.24: RSL definition for family Sequencer [BKK10]



with initial memory snapshot $\{x \mapsto 0, y \mapsto 1\}$

Figure 3.25: Constraint automaton for the OddFibonacci$_2$ protocol. The producer has access to port A, one consumer has access to port B, the other consumer has access to port C, and the producers and the consumer use buffers x and y for temporary storage of data.

tions in the previous subsection (for the denotational semantics of `LateAsyncMerger2(A,B;C)` and `LateAsyncMerger(A[1..2];C)`), without having to actually compute the corresponding constraint automaton (which may demand substantial computational resources). Eventually, a FOCAML compiler may still have to compute this corresponding constraint automaton, but I decouple that effort from FOCAML interpretation. If the FOCAML interpreter succeeds in interpreting an instantiated family signature, it displays a list of the primitives denoted by that signature in a designated view in Eclipse. The FOCAML interpreter also has a rather ad-hoc type checker (not formalized), primarily to provide meaningful error messages to users.

The editor/parser/interpreter plugin for FOCAML can also translate Reo circuits—formatted by using the ECT plugins for Eclipse (see `http://reo.project.cwi.nl`), which constitute an IDE for Reo—into FOCAML code. This translation makes all compilation techniques presented in this thesis directly applicable to Reo as well, not only in theory but also in practice.

Example I: OddFibonacci

Suppose that I must write a program that consists of a producer and k consumers. My protocol specification states that the producer sends its data to the consumers synchronously but "predictably unreliably" in the following sense. After the i-th send of the producer, the consumers synchronously receive the i-th Fibonacci number, denoted by $fib(i)$, if $fib(i) \bmod 2 = 1$. Otherwise, if $fib(i) \bmod 2 = 0$, the consumers do not receive anything (and their pending operations remain pending). Either way, the datum originally sent by the producer never reaches the consumers and gets lost in communication. The OddFibonacci family of constraint automata for this protocol has one natural number parameter, for k. Although its practical use may seem questionable, OddFibonacci well-illustrates the expressiveness of constraint automata and serves as a useful example later on.

Figure 3.25 shows a member of the OddFibonacci$_2$ subfamily; Figure 3.26 shows a circuit for that same member; Figure 3.27 shows a FOCAML definition for the entire OddFibonacci family and a main definition for the same member as in the previous figures. Some clarifications and remarks:

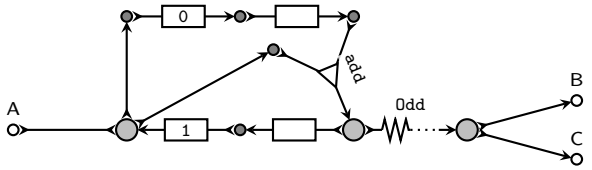


Figure 3.26: Circuit for a member of subfamily OddFibonacci$_2$

```
1  Replicator(in;out[]) = {
2    let k = #out {
3      if (k == 1) {
4        Sync(in;out[1])
5      } else if (k == 2) {
6        Replicator2(in;out[1],out[2])
7      } else {
8        Replicator2(P[2];out[1],out[2])
9          mult { prod i:3..k-1 { Replicator2(P[i];P[i-1],out[i]) } }
10           mult Replicator2(in;P[k-1],out[k])
11 } } }

12 OddFibonacciPart(a,c;f,h) = {
13   Fifo(a;B)
14   mult Sync(c;D)
15     mult BinOp<'add'>(B,D;E)
16       mult Replicator2(E;f,G)
17         mult Filter<'Odd'>(G;h)
18 }

19 OddFibonacci(in;out[]) = {
20   OddFibonacciPart(A,C;F,H)
21     mult Fifo(F;P1)
22       mult FifoFull<'1'>(P1;P2)
23         mult Replicator2(P2;C,P3)
24           mult Replicator2(P3;P4,P5)
25             mult SyncDrain(P4,in;)
26             mult Sync(P5;P6)
27               mult FifoFull<'0'>(P6;A)
28     mult Replicator(H;P7[1..#out])
29       mult { prod i:1..#out { Sync(P7[i];out[i]) } }
30 }

31 main = { OddFibonacci(A;[B,C]) }
```

Figure 3.27: FOCAML definitions for families Replicator, OddFibonacciPart, and OddFibonacci, and a main definition for a member of OddFibonacci$_2$

- The extralogical symbols in the data constraints in Figure 3.25 have the following meaning. Data term $\texttt{add}(x_1, x_2)$ evaluates to $x_1 + x_2$. Data relation $\texttt{Odd}(x)$ holds true iff $x \bmod 2 = 1$.
- Memory cells x and y in the constraint automaton in Figure 3.25 gener-

ally contain $fib(i-2)$ and $fib(i-1)$. The first line of their data constraint guarantees that both transitions in this automaton update the datum in x to the datum previously in y (after which x contains $fib(i-1)$), while they update the datum in y to the sum of the data previously in x and y (after which y contains $fib(i)$). The left transition permits instances of interaction where $fib(i) \bmod 2 = 1$ holds true, in which case (i) the producer and the consumers synchronize and (ii) the consumers receive the i-th Fibonacci number. The right transition permits instances of interaction where $fib(i) \bmod 2 = 0$ holds true, in which case the producer does not communicate with the consumers whatsoever (even though the `put` of the producer succeeds from the perspective of the producer, who never knows that its datum never reaches the consumers).

- In Figure 3.27, the purpose of separating `OddFibonacciPart(a,c;f,h)` from `OddFibonacci(in;out[1..k])`, which may seem rather arbitrary—or even pointless—at this point, becomes clear in Chapter 6.
- The instantiated family signature `OddFibonacci(A;[B,C])` has a denotation behaviorally equivalent to the constraint automaton in Figure 3.25 but not structurally equal. In particular, I simplified the data constraints in the denotation of `OddFibonacci(A;[B,C])`, for the sake of clarity and presentation, to obtain the constraint automaton in Figure 3.25.

Example II: Chess

The following example originates from a discussion with Kasper Dokter, a close colleague and fellow PhD student at CWI.

Kasper participates in a chess competition. After every game, Kasper uses special chess programs, called *chess engines*, to help him analyze his play. Chess engines algorithmically try to find the best move in a certain input *position* (i.e., a state of the game board). Unfortunately, chess has a huge state space, and chess engines typically lack the resources to exhaustively explore this whole space. Instead, through heuristics and user-controllable search parameters, chess engines usually use their limited resources to make a best-effort approximation of the theoretically best move. Different chess engines support different such heuristics and parameters. Consequently, different chess engines may find different approximations of the theoretically best move when presented the same input position.

Kasper wanted to write a program that automatically invokes a number of different chess engines on the same position, compares their results, perhaps analyzes the different results some more, and repeats this process for the successor positions corresponding to the computed best moves. Specifically, Kasper wanted to implement the protocol specification of this program using Reo. This ambition caught my interest, because the work involved seemed quite nontrivial. In the end, Kasper's chess program inspired me to write my own chess program, whose protocol includes the same elements that made

Kasper's original chess program interesting to me. The set of workers in my program consists of k chess engines for "Team White", one chess engine for "Team Black", and a display. With the right complementary protocol, my program simulates a game of chess between Team White and Team Black, where the display shows the current position on a virtual game board on the screen. Next, I discuss what comprises this protocol.

The $k + 1$ chess engines in my program use at least two buffers to interact with each other, namely to store a *history* of the moves played so far (which effectively represents the current position of the game). Depending on whether Team White or Team Black played the last move, exactly one of these "history buffers" contains an up-to-date history. Initially, the "black history buffer" (for when Team Black played the last move) contains the empty history. In the first instance of interaction, the k chess engines for Team White synchronously receive the empty history from the black history buffer. Subsequently, these chess engines evaluate this sequence to an actual position and try to find a best move. Afterward, these chess engines synchronously send their raw output into their environment. This environment takes care of parsing this raw output (i.e., extracting the best move), combining the proposed best moves into one definite move for Team White (e.g., by majority vote), delivering this move to the display, appending this move to the previous move history, and storing the updated move history in the white history buffer. All this, including the sends by the chess engines, occurs synchronously, as part of one atomic instance of interaction. After the previous two instances of interaction involving Team White, two similar such instances occur for Team Black, except that Team Black consists of only one chess engine, which makes some steps unnecessary. Afterward, the black history buffer contains a datum again, and the whole process repeats itself until the game ends. At any time during the game, whenever the display receives a new move, it updates the screen accordingly. The Chess family of constraint automata for this protocol has one natural number parameter, for k.

Figure 3.28 shows a member of the Chess_3 subfamily; Figure 3.29 shows a circuit for that same member; Figure 3.30 shows a FOCAML definition for the entire Chess family and a main definition for the same member as in the previous figures.

The extralogical symbols in the data constraints in Figure 3.28 have the following meaning. Data term $\texttt{parse}(x)$ evaluates to a string representation of the best move in raw output x. Data term $\texttt{concatenate}(x_1, x_2)$ evaluates to the concatenation of x_1 and x_2. For instance, $\texttt{concatenate}(\texttt{"d2d4 f7f5"}, \texttt{"c2c4"})$ evaluates to `"d2d4 f7f5 c2c4"` (this string, suggested to me for inclusion in this thesis by Kasper, describes a chess opening called the Dutch Defense). Data term $\texttt{majority}(x)$ evaluates to the space-separated substring in x with the most occurrences in x. For instance, $\texttt{majority}(\texttt{"d2d4 d2d4 e2e4"})$ evaluates to `"d2d4"`. Data relation $\texttt{Move}(x)$ holds true iff x represents a valid move (in which case the game has not finished yet).

As in the previous subsection, and for the same reason, the instantiated family signature

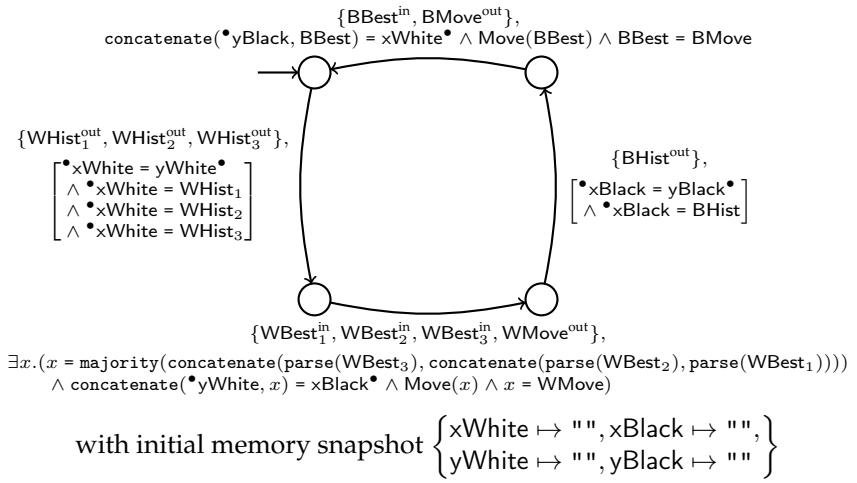


Figure 3.28: Constraint automaton for the Chess_3 protocol. For $i \in \{1, 2, 3\}$, every chess engine i for Team White has access to port WHist_i and to port WBest_i, the chess engine for Team Black has access to port BHist and port BBest, the display has access to ports WMove and BMove, and the chess engines use buffers xWhite, xBlack, yWhite, and yBlack for temporary storage of data.

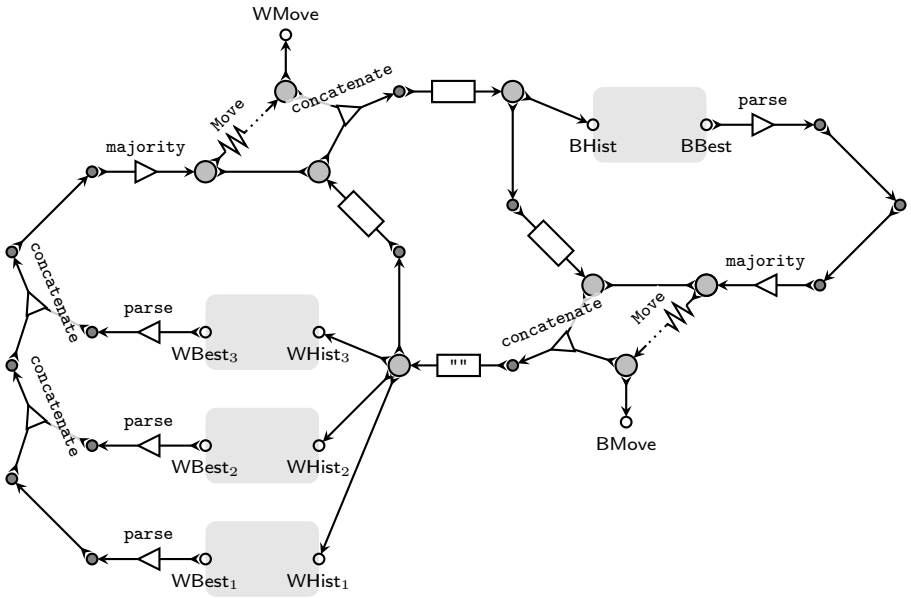


Figure 3.29: Circuit for a member of subfamily Chess_3, where gray boxes represent chess engines

```
1  Concatenator(in[];out) = {
2    Sync(in[1];P[1])
3      mult { prod i:1..#in-1 { BinOp<'concatenate'>(in[i+1],P[i];P[i+1]) } }
4        mult Sync(P[#in];out)
5  }

6  Team(in,best[];hist[],move,out) = {
7    Replicator2(in;P1,P5)
8      mult Fifo(P1;P2)
9        mult Replicator2(P2;P3,P4)
10     mult Replicator(P5;hist[1..#hist])
11   mult { prod i:1..#best { Transformer<'parse'>(best[i];P6[i]) } }
12     mult Concatenator(P6[1..#best];P7)
13       mult Transformer<'majority'>(P7;P8)
14         mult Replicator2(P8;P9,P10)
15           mult SyncDrain(P3,P9;)
16           mult Filter<'Move'>(P10;P11)
17             mult Replicator2(P11;P12,P13)
18               mult Sync(P12;move)
19               mult BinOp<'concatenate'>(P4,P13;out)
20 }

21 Chess(white_best[],black_best;white_hist[],black_hist,white_move,black_move) = {
22   Team(
23     white_in,white_best[1..#white_best];
24     white_hist[1..#white_hist],white_move,white_out
25   )
26     mult Fifo(white_out;black_in)
27       mult Team(black_in,[black_best];[black_hist],black_move,black_out)
28         mult FifoFull<'"'">(black_out;white_in)
29 }

30 main = {
31   Chess([WBest1,WBest2,WBest3],BBest;[WHist1,WHist2,WHist3],BHist,WMove,BMove)
32 }
```

Figure 3.30: FOCAML definitions for families Concatenator, Team, and Chess, and a main definition for a member of Chess_3

```
Chess(
  [WBest1,WBest2,WBest3],BBest;
  [WHist1,WHist2,WHist3],BHist,WMove,BMove
)
```

has a denotation behaviorally equivalent to the constraint automaton in Figure 3.28 but not structurally equal.

Example III: NAS Parallel Benchmarks

In the late 1980s and early 1990s, the *NASA Advanced Supercomputing* (NAS) Division—then called the *Numerical Aerodynamic Simulation* (NAS) Program—at NASA Ames Research Center faced a "grand challenge" [BBB+94]: "to ad-

vance the state of computational aerodynamics" and "to provide the Nation's aerospace research and development community by the year 2000 a high-performance, operational computing system capable of simulating an entire aerospace vehicle system within a computing time of one to several hours". The development of new supercomputing technology for large-scale parallel processing seemed imperative to the successful completion of this challenge. At some point, as part of this program, researchers at NASA Ames realized that "benchmarking and performance evaluation of [highly parallel systems] has not kept pace with advances in hardware, software and algorithms" and, specifically, that "there is as yet no generally accepted benchmark program or even a benchmark strategy for these systems". Filling this void, Bailey et al. developed a new set of benchmark specifications for evaluating the performance of highly parallel system, derived from *computational fluid dynamics* (CFD) applications, called the *NAS Parallel Benchmarks* (NPB) [BBB+91, BBB+94]. NPB has become "a popular set of kernels and applications used for supercomputer evaluation" [HP11c]. In fact, already ten years after its release, the *high-performance computing* (HPC) community had accepted NPB as "an instrument for evaluating performance of parallel computers, compilers, and tools" [FSJY02], having become "a standard indicator of computer performance".

The first implementations of NPB consisted of C and Fortran code, using MPI; later, Jin et al. wrote another implementation, using OpenMP [JFY99]. In the early 2000s, as interest in Java by the HPC community increased, Frumkin et al. derived an implementation of NPB in Java from the existing OpenMP version [FSJY02, FSJY03]. Because the compiler that I present in later chapters generates Java code, this Java implementation of NPB seems an interesting reference point against which to evaluate the performance of Java code compiled from FOCAML programs for NPB. Moreover, because the NPB suite consists of full programs—instead of just protocols—experiments with NPB provide a complementary perspective on the performance of my compiler-generated code, beside experiments that focus exclusively on protocols alone.

The Java implementation of NPB consists of seven benchmarks: four computational *kernels* (this implementation of NPB excludes its fifth "embarrassingly parallel" kernel) and three simulated CFD applications. The four computational kernels represent common numerical methods in CFD applications. More specifically [BBB+94]:

- NPB-FT (Fourier transform)
  Benchmark that computes the solution of a partial differential equation, using the forward and inverse Fast Fourier Transform algorithm.

- NPB-MG (multigrid)
  Benchmark that computes an approximate solution u to the discrete Poisson problem $\nabla^2 u = v$, using the V-cycle multigrid algorithm.

- NPB-CG (conjugate gradient)
  Benchmark that computes an estimate of the largest eigenvalue of a sym-

metric positive definite sparse matrix with a random pattern of nonzeros, using the conjugate gradient algorithm.

- NPB-IS (integer sorting)
  Benchmark that computes a sorted list of uniformly distributed integer keys, using a histogram-based integer sorting algorithm.

Beside these four computational kernels, the three simulated CFD applications compute the solution of a synthetic system of nonlinear partial differential equations, using techniques similar to those used in real CFD applications. Contrasting real CFD applications, however, these benchmarks lack pre- and postprocessing and have no disk I/O. As such, these benchmarks constitute stripped-down, but still representative, versions of real CFD applications, simulating both their interaction and computation aspects. These three benchmarks, called NPB-BT (block tridiagonal systems of equations), NPB-SP (scalar pentadiagonal systems), and NPB-LU (lower and upper triangular systems), differ in their algorithm for solving Navier-Stokes equations [BBB+94].

Every benchmark in the Java implementation of NPB consists of one *master* and k *slaves*. In each of these benchmarks, initially, all slaves wait for their master to dispatch work. Subsequently, every slave starts performing its assigned work, while the master waits until it has received a signal from every slave about the completion of its work. Thus, all benchmarks incorporate the classical *master–slaves interaction pattern*, implemented in Java with invocations to methods `wait` and `notify` for monitor-based synchronization and shared memory for data communication. Figure 3.31 shows Java code for this pattern. Note that the master dispatches the same work, here represented by an `Object`, to every worker in the same iteration of the main loop (i.e., the master invokes `newWork` outside the inner loop). This `Object` contains information specific to the current iteration of the main loop that the workers should be aware of when performing their computation. Additionally, the workers access global data structures in shared memory (e.g., an array partitioned into per-worker subarrays and distributed among the workers during initialization).

In all but one benchmarks, slaves interact only with their master and never with each other (i.e., they have no dependencies among them). In those benchmarks, thus, the previous master–slaves interaction pattern covers all instances of interaction. In benchmark NPB-LU, in contrast, the slaves additionally have pipelined dependencies between them. Frumkin et al. ensure the satisfaction of these dependencies in their implementation by using a *relay-race interaction pattern*. In this pattern, the master dispatches work to all its slaves but waits only for the last one to finish (according to some total order on the slaves). Meanwhile, every slave waits for a signal from its predecessor before it starts performing its assigned work (possibly in a number of steps), except for the first slave, who immediately starts. Once a slave completes (a step of) its work, it sends a signal to its successor, except for the last slave, who signals the master. Figure 3.32 shows Java code for this pattern.

I took the Java implementation of NPB as my starting point for developing

```
1  public class abstract Master extends Thread {
2    public volatile Slave[] slaves;
3
4    protected abstract Object newWork();
5
6    public void run() {
7      while (true) {
8        Object work = newWork();
9        for (int i = 0; i < slaves.length; i++) {
10         slaves[i].work = work;
11         slaves[i].done = false;
12         synchronized (slaves[i]) {slaves[i].notify()}
13       }
14       synchronized (this) {
15         for (int i = 0; i < slaves.length; i++)
16           while (!slaves[i].done)
17             try {wait();} catch (InterruptedException exc) {}
18 } } } }

19 public abstract class Slave extends Thread {
20   public volatile Master master;
21   public volatile Object work;
22   public volatile boolean done = true;
23
24   protected abstract void work(); // accesses global data structures in shared memory
25
26   public void run() {
27     while (true) {
28       synchronized (this) {
29         while (done)
30           try {wait();} catch (InterruptedException exc) {}
31       }
32       work();
33       done = true;
34       synchronized (master) {master.notify();}
35 } } }
```

Figure 3.31: Java code for the master–slaves interaction pattern [FSJY02]

a FOCAML implementation of NPB. First, I isolated all instances of the master–slaves interaction pattern (in all benchmarks) and the relay-race interaction pattern (in NPB-LU). I subsequently rewrote the Java code for those patterns from their previous *monitor-based* versions in Figures 3.31 and 3.32 into *port-based* versions in Figure 3.33, thereby effectively "factoring out" all interaction code from the original codebase. (These figures constitute one of the rare exceptions in this thesis, where I qualify ports as "input" or "output" from the perspective of workers.) Beside the modifications in Figure 3.33, I also modified the constructors of masters and slaves (primarily adding `InputPort` and `OutputPort` parameters), and some associated initialization code. Mainly, this added initialization code distributes *references* to large global data structures in shared memory among the workers, *as values*. In principle, the compiler should automatically infer when to substitute reference-passing for value-passing, but

```
public abstract class RelayRaceMaster extends Master {
  public void run() {
    while (true) {
      Object work = newWork();
      for (int i = 0; i < slaves.length; i++) {
        slaves[i].work = work;
        slaves[i].done = false;
        synchronized (slaves[i]) {slaves[i].notify()}
      }
      synchronized (this) {
        while (!slaves[slaves.length].done)
          try {wait();} catch (InterruptedException exc) {}
} } } }

public abstract class RelayRaceSlave extends Slave {
  public volatile int todo = 0;
  public volatile int nSteps;
  public volatile int id;

  public void run() {
    while (true) {
      synchronized (this) {
        while (done)
          try {wait();} catch (InterruptedException exc) {}
        for (int i = 0; i < nSteps; i++) {
          if (id > 0)
            while (todo == 0)
              try {wait();} catch (InterruptedException exc) {}
          work();
          todo--;
          if (id < master.slaves.length - 1)
            synchronized (master.slaves[id + 1]) {
              master.slaves[id + 1].todo++;
              master.slaves[id + 1].notify();
        }   }
        done = true;
        if (id == master.slaves.length - 1)
          synchronized (master) {master.notify();}
} } } }
```

Figure 3.32: Java code for the relay-race interaction pattern [FSJY02]

as I explained in Chapter 1, such optimization techniques lie beyond my current scope; see the MSc thesis of Van de Nes [vdN15]. Once I had completed all these modifications, I only needed to write FOCAML family definitions for the protocols in the two interaction patterns, plus a main definition for every benchmark (to hook the Java code into the FOCAML code).

Before turning to the FOCAML code, note the following: `PortBasedRelayRaceSlave` contains no information whatsoever about the relation between, on the one hand, `Ports X` and `Y` and, on the other hand, their connected slaves. In contrast, the monitor-based code for the relay-race interaction pattern in Figure 3.32 explicitly encodes the fact that a slave depends on its predecessor and successor. By factoring out the interaction code from the original code-

```
1  public abstract class PortBasedMaster extends Master {
2    public volatile OutputPort A;
3    public volatile InputPort B;
4
5    public void run() {
6      while (true) {
7        Object work = newWork();
8        for (int i = 0; i < slaves.length; i++)
9          A.putUninterruptibly(work);
10       for (int i = 0; i < slaves.length; i++)
11         B.getUninterruptibly();
12 } } }

13 public abstract class PortBasedSlave extends Slave {
14   public volatile OutputPort A;
15   public volatile InputPort B;
16
17   public void run() {
18     while (true) {
19       B.getUninterruptibly();
20       work();
21       A.putUninterruptibly(new Object());
22 } } }

23 public abstract class PortBasedRelayRaceMaster extends PortBasedMaster {
24   public void run() {
25     while (true) {
26       Object work = newWork();
27       for (int i = 0; i < slaves.length; i++)
28         A.putUninterruptibly(work);
29       B.getUninterruptibly();
30 } } }

31 public abstract class PortBasedRelayRaceSlave extends PortBasedSlave {
32   public volatile int nSteps;
33   public volatile int id;
34   public volatile OutputPort X;
35   public volatile InputPort Y;
36
37   public void run() {
38     while(true) {
39       B.getUninterruptibly();
40       for (int i = 0; i < nSteps; i++) {
41         if (id > 0)
42           X.getUninterruptibly();
43         work();
44         if (id < master.slaves.length - 1)
45           Y.putUninterruptibly(new Object());
46       }
47       if (id == master.slaves.length)
48         A.putUninterruptibly(new Object());
49 } } }
```

Figure 3.33: Java code for the master–slaves and relay-race interaction patterns based on ports (cf. Figures 3.31 and 3.32)

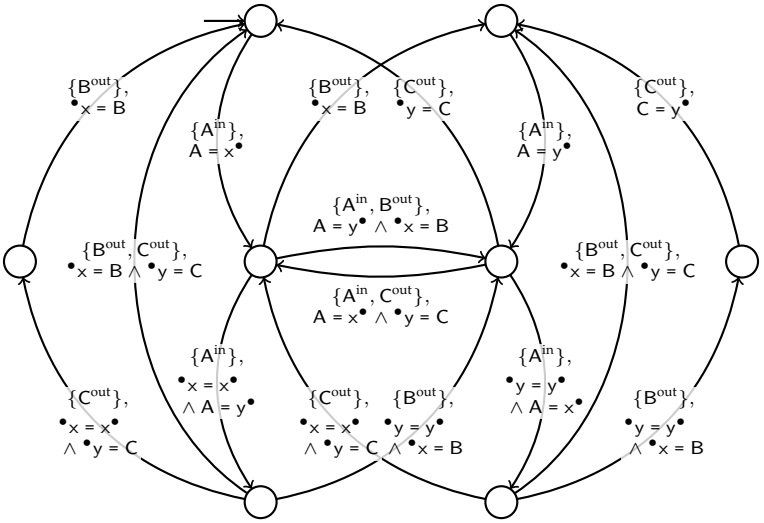
base, I pushed the information about interaction between slaves to the FOCAML implementation of the corresponding protocol specification, as demonstrated shortly. `PortBasedRelayRaceSlave`, then, illustrates a key point of using a separate DSL for interaction: worker subprograms in a complementary GPL contain no information about interaction or protocols, making both worker and protocol subprograms simpler to write and reason about and more reusable.

To model the master–slaves interaction pattern using constraint automata, I break this pattern down into two constituent protocols: one for the master to dispatch work to its slaves and one for the slaves to signal their master about the completion of their work. For each of these two protocols, I define *two* families of constraint automata, with—technically speaking—behaviorally nonequivalent members (the purpose of which I explain shortly). This yields a total of four families: MasterToSlavesA and SlavesToMasterA, whose members rather literally correspond to the Java code in Figure 3.31, and MasterToSlavesB and SlavesToMasterB, whose members correspond to that code less literally but nevertheless respect the *intention* behind the master–slaves interaction pattern. Each of these families has a natural number parameter for their number of slaves. Members of MasterToSlavesA and SlavesToMasterA straightforwardly compose into members of the MasterSlavesInteractionPatternA family, which comprehensively models the master–slaves interaction pattern; the same holds true of MasterToSlavesB, SlavesToMasterB, and MasterSlavesInteractionPatternB.

Figures 3.34 and 3.35 show members of the $\mathsf{MasterToSlavesA}_2$ and $\mathsf{SlavesToMasterA}_2$ subfamilies; Figure 3.36 shows circuits for those same members; Figure 3.37 shows FOCAML definitions for the entire MasterToSlavesA family, the entire SlavesToMasterA family, the entire MasterSlavesInteractionPatternA family, and a main definition for a member of the latter family corresponding to the multiplication of the same members as in the previous figures. As in the previous subsections, and for the same reasons as there, the instantiated family signatures `MasterToSlavesA(A;[B,C])` and `SlavesToMasterA([D,E];F)` have a denotation behaviorally equivalent to the constraint automata in Figures 3.34 and 3.35 but not structurally equal.

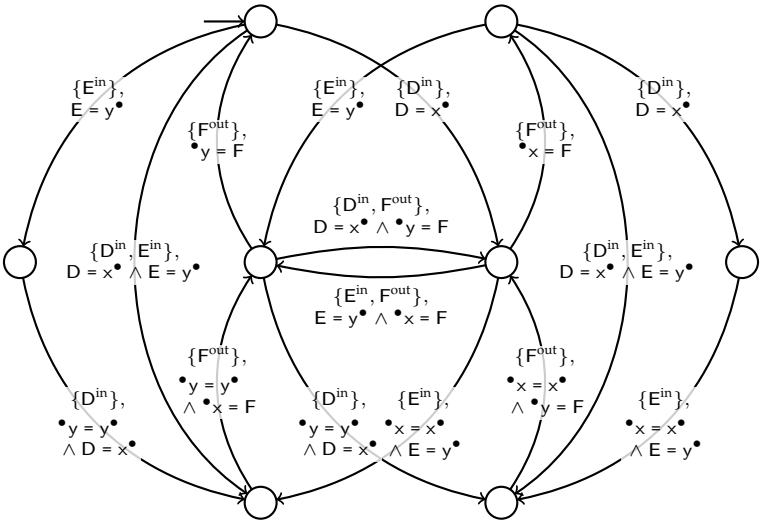
Members of MasterSlavesInteractionPatternA enforce an order in which the master signals its slaves and vice versa, achieved through members of Sequencer. The Java code for the master–slaves interaction pattern in Figure 3.31 also enforces such an order, through Java's fixed iteration order in loops. I use Fifos just before the output ports in members of MasterToSlavesA and just after the input ports in members of SlavesToMasterA to make the interaction between a master and its slaves asynchronous. This corresponds to the nonblocking semantics of `notify` in the Java code (i.e., a `notify` invocation immediately returns; it does not go to sleep until another thread invokes `wait`). As such, members of MasterSlavesInteractionPatternA rather literally correspond to the Java code in Figure 3.31.

The constraint automaton in Figure 3.34 furthermore shows that, in principle, the MasterToSlavesA protocol admits the following questionable sequence of interaction: (i) the master `puts` on A to dispatch work to the slave on B, (ii) the master `puts` on A to dispatch work to the slave on C, (iii) the slave on B



with initial memory snapshot $\{x \mapsto 0, y \mapsto 0\}$

Figure 3.34: Constraint automaton for the $\mathsf{MasterToSlavesA}_2$ protocol. The master has access to port A, one slave has access to port B, the other slave has access to port C, and the master and the slaves use buffers x and y for temporary storage of data.



with initial memory snapshot $\{x \mapsto 0, y \mapsto 0\}$ and

Figure 3.35: Constraint automaton for the $\mathsf{SlavesToMasterA}_2$ protocol. One slave has access to port D, the other slave has access to port E, the master has access to port F, and the slaves and the master use buffers x and y for temporary storage of data.

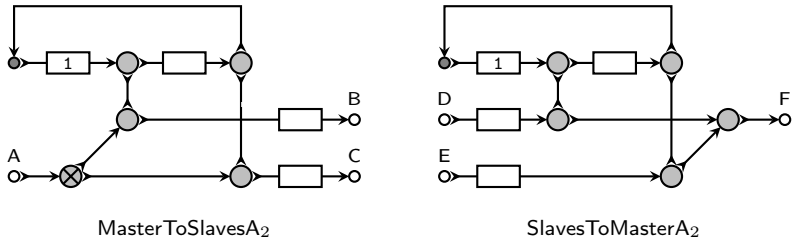


Figure 3.36: Circuits for members of subfamilies MasterToSlavesA$_2$ and SlavesToMasterA$_2$

```
50 MasterToSlavesA(in;out[]) = {
51   let n = #out {
52     Sync(in;P1)
53       mult Router(P1;P2[1..n])
54         mult { prod i:1..n { Sync(P2[i];P3[i]) } }
55           mult { prod i:1..n { Replicator2(P3[i];P4[i],P5[i]) } }
56             mult Sequencer(P4[1..n];)
57             mult { prod i:1..n { Fifo(P5[i];out[i]) } }
58 } }

59 SlavesToMasterA(in[];out) = {
60   let n = #in {
61     { prod i:1..n { Fifo(in[i];P1[i]) } }
62       mult { prod i:1..n { Replicator2(P1[i];P2[i],P3[i]) } }
63         mult Sequencer(P2[1..n];)
64         mult { prod i:1..n { Sync(P3[i];P4[i]) } }
65           mult Merger(P4[1..n];P5)
66             mult Sync(P5;out)
67 } }

68 MasterSlavesInteractionPatternA(
69   master_in,slaves_in[];
70   slaves_out[],master_out
71 ) = {
72   MasterToSlavesA(master_in;slaves_out[1..#slaves_out])
73   mult SlavesToMasterA(slaves_in[1..#slaves_in];master_out)
74 }

75 main = MasterSlavesInteractionPatternA(A,[D,E];[B,C],F)
```

Figure 3.37: FOCAML definitions for families MasterToSlavesA, SlavesToMasterA, and MasterSlavesInteractionPatternA, and a main definition for a member of MasterSlavesInteractionPatternA$_2$

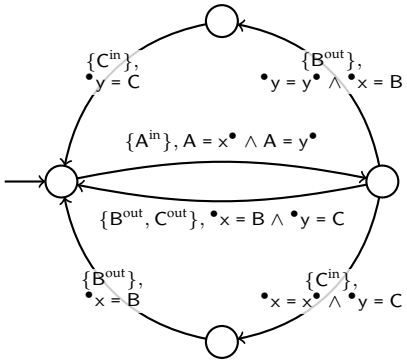


Figure 3.38: Constraint automaton for the MasterToSlavesB$_2$ protocol. The master has access to port A, one slave has access to port B, the other slave has access to port C, and the master and the slaves use buffers x and y for temporary storage of data.

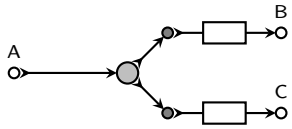


Figure 3.39: Circuit for a member of subfamily MasterToSlavesB$_2$

gets on B to receive its assigned work, (iv) the master puts on A to dispatch more work to the slave on B, and (v) the slave on B gets for the second time on B, presumably after having finished its first piece of work, but before the slave on C has received anything. Considered in isolation, the interaction code in Figure 3.31 (i.e., `wait`, `notify`, and **synchronized**) admits the same sequence of interaction, and so, also in this respect, MasterSlavesInteractionPatternA rather literally corresponds to the code in that figure. In practice, however, the previous sequence of interaction *never* occurs—neither in the original monitor-based code nor in my modified port-based code—because the complementary computation code provides additional guarantees. Notably, the master waits for a signal of every slave after it has dispatched work. Although I can express such guarantees directly in FOCAML code, thereby embedding them in the protocol, I do not pursue that in MasterSlavesInteractionPatternA, which I want to keep as close to the Java code as possible for a fairer comparison.

Figure 3.38 show a member of the MasterToSlavesB$_2$ subfamily; Figure 3.39 shows a circuit for that same member; Figure 3.40 shows FOCAML definitions for the entire MasterToSlavesB family, the entire SlavesToMasterB family, the entire MasterSlavesInteractionPatternB family, and a main definition for a member of the latter family corresponding to the multiplication of the same member as in the previous figures and the constraint automaton in Figure 3.1. The latter figure shows a member of the SlavesToMasterB$_2$ subfamily, which equals the EarlyAsyncMerger$_2$ subfamily; Figure 3.2 shows a circuit for that same member. As before, and for the same reasons as there, the instantiated family signatures `MasterToSlavesB(A;[B,C])` and `SlavesToMasterB([D,E];F)` have a denota-

```
1  MasterToSlavesB(in;out[]) = {
2    let n = #out {
3      Sync(in;P1)
4        mult Replicator(P1;P2[1..n])
5          mult { prod i:1..n { Sync(P2[i];P3[i]) } }
6            mult { prod i:1..n { Fifo(P3[i];out[i]) } }
7  } }

8  SlavesToMasterB(in[];out) = { EarlyAsyncMerger(in[1..#in];out) }

9  MasterSlavesInteractionPatternB(
10   master_in,slaves_in[];
11   slaves_out[],master_out
12 ) = {
13   MasterToSlavesB(master_in;slaves_out[1..#slaves_out])
14   mult SlavesToMasterB(slaves_in[1..#slaves_in];master_out)
15 }

16 main = MasterSlavesInteractionPatternB(A,[D,E];[B,C],F)
```

Figure 3.40: FOCAML definitions for families MasterToSlavesB, SlavesToMasterB, and MasterSlavesInteractionPatternB, and a main definition for a member of MasterSlavesInteractionPatternB$_2$

tion behaviorally equivalent to the constraint automata in Figures 3.38 and 3.1 but not structurally equal.

Contrasting members of MasterSlavesInteractionPatternA, members of MasterSlavesInteractionPatternB enforce no order in which the master sends signals to its slaves and vice versa. Instead, using MasterSlavesInteractionPatternB, the master dispatches work to all its slaves at the same time, achieved through members of Replicator. To justify this act of eliminating the order that Frumkin et al. impose in their Java code for the master–slaves interaction pattern in Figure 3.31 observe that this order hardly seems an intentional part of their protocol specification but rather, an artifact of using Java: Frumkin et al. simply cannot express in Java that the iteration order of a loop does not matter. As such, Frumkin et al. had no choice but to "overimplement" their protocol specification, and in the process, make their true intention practically impossible to retrieve by the Java compiler. After all, how can the Java compiler make a similar "soft" analysis as mine to determine that Frumkin et al. not truly intended to impose an order but that the Java language simply forced them to do so? This question matters from a performance point of view, because it seems quite reasonable to assume that eliminating the order leads to better performance (or at least not worse); actual experiments follow in this thesis. In FOCAML, contrasting Java, I can express that the master dispatches work to all its slaves instantaneously, synchronously, atomically, by using Replicator. By doing so, I avoid overimplementing the protocol specification, leaving room for the FOCAML compiler to decide whether or not imposing an order makes sense from a performance point of view. Moreover, using MasterSlaves-

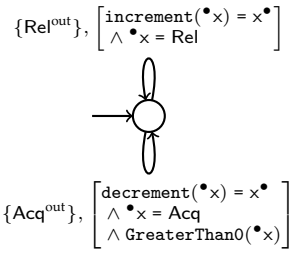


Figure 3.41: Constraint automaton for the Semaphore protocol.
One slave has access to port Acq, the other slave has access to port Rel, and the slaves use buffer x for temporary storage of data.

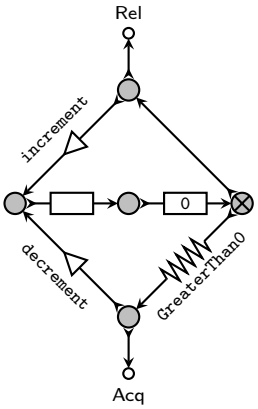


Figure 3.42: Circuit for a member of family Semaphore

InteractionPatternB, the master needs to perform only one `put` to dispatch work instead of k, for k slaves; this further simplifies the code in Figure 3.33.

To model the relay-race interaction pattern using constraint automata, I break this pattern down into three constituent protocols: one for the master to dispatch work to its slaves (as in the master–slaves interaction pattern), one for the slaves to signal their neighbors, and one for the last slave to signal its master about the completion of all the work. For the first of these three protocols, I reuse families MasterToSlavesA and MasterToSlavesB; for the third of these three protocols, I use family Fifo. For the second of these three protocols, I define the Semaphore family of constraint automata. Every member of Semaphore effectively behaves as a classical semaphore between one "releasing worker" and one "acquiring worker". As before, MasterToSlavesA and MasterToSlavesB have a natural number parameter for their number of slaves. The number of slaves also determines the number of Semaphores necessary to implement the relay-race interaction pattern. Members of MasterToSlavesA, Semaphore, and Fifo straightforwardly compose into members of the RelayRaceInteractionPatternA family, which comprehensively models the relay–race interaction pattern; the same holds true of MasterToSlavesB, Semaphore, Fifo, and RelayRaceInteractionPatternB.

Figure 3.41 shows a member of the Semaphore family; Figure 3.42 shows a circuit for that same member; Figure 3.37 shows FOCAML definitions for the entire Semaphore family, the entire RelayRaceInteractionPatternA family, and the entire RelayRaceInteractionPatternB family. Finally, Figure 3.44 shows a circuit for the multiplication of a member of MasterToSlaves_2 and a member of

```
Semaphore(;acq,rel) = {
  Fifo(P1;P2)
    mult FifoFull<'0'>(P2;P3)
      mult Router2(P3;P4,P9)
        mult Blocker<'GreaterThan0'>(P4;P5)
          mult Replicator2(P5;P6,P7)
            mult Sync(P6;acq)
            mult Transformer<'decrement'>(P7;P8)
        mult Sync(P9;P10)
          mult Replicator2(P10;P11,P12)
            mult Sync(P11;rel)
            mult Transformer<'increment'>(P12;P13)
              mult Merger2(P8,P13;P1)
}

RelayRaceInteractionPatternA(
  master_in,last_slaves_in;
  slaves_out[],master_out,slaves_acq[],slaves_rel[]
) = {
  let n = #slaves_out {
    MasterToSlavesA(master_in;slaves_out[1..#slaves_out])
    mult { prod i:1..n { Semaphore(;slaves_acq[1+(i%n)],slaves_rel[i]) } }
    mult Fifo(last_slaves_in;master_out)
} }

RelayRaceInteractionPatternB(
  master_in,last_slaves_in;
  slaves_out[],master_out,slaves_acq[],slaves_rel[]
) = {
  let n = #slaves_out {
    MasterToSlavesB(master_in;slaves_out[1..#slaves_out])
    mult { prod i:1..n { Semaphore(;slaves_acq[1+(i%n)],slaves_rel[i]) } }
    mult Fifo(last_slaves_in;master_out)
} }
```

Figure 3.43: FOCAML definitions for families Semaphore, RelayRaceInteractionPatternA, and RelayRaceInteractionPatternB

Semaphore. The extralogical symbols in the data constraints in Figure 3.41 have the following obvious meaning: data term `increment`(x) evaluates to $x + 1$, data term `decrement`(x) evaluates to $x-1$, while data relation `GreaterThan0`(x) holds true iff $x > 0$.

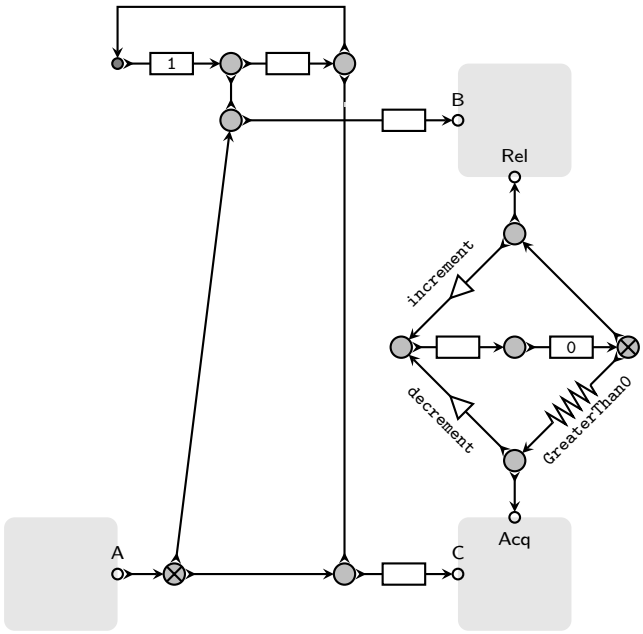


Figure 3.44: Circuit for the multiplication of a member of subfamily MasterToSlavesA_2 and a member of family Semaphore, where gray boxes represent the master (left) and its two workers (right)

Chapter 4

Basic Compilation

In Chapters 2 and 3, I presented the semantics and syntax of FOCAML. In this chapter, I continue my presentation of FOCAML with the introduction of its first compiler, called *Lykos*. This name reflects my "reverse approach" of starting from constraint automata and later adopting Reo as a syntax as opposed to starting from Reo and later adopting constraint automata as a semantics, as follows:

$$\text{lykos} \xrightarrow{\alpha\beta\gamma\cdots} \lambda\upsilon\kappa o\varsigma \xrightarrow{\text{english}} \text{wolf} \xrightarrow{\text{reverse}} \text{flow} \xrightarrow{\text{greek}} \rho\epsilon o \xrightarrow{abc\cdots} \text{reo}$$

In Section 4.1, I present the principles behind basic FOCAML compilation. In particular, I discuss two opposite compilation approaches: one that yields maximally parallel code (with high throughput, at the cost of high latency) and another that yields maximally sequential code (with low latency, at the cost of low throughput). In Section 4.2, I present Lykos. Lykos generates Java code under the second compilation approach. I also give code examples and conclude with some experimental results on performance.

4.1 Theory

*(With Arbab, I previously published fragments of the material in this section in workshop papers [**J**A13b, **J**A14] and in a journal paper [**J**A16].)*

Basics

Recall from Chapter 1 that using a DSL for interaction, software engineers write their worker subprograms in a GPL and their protocol/main subprograms in a complementary DSL; a compiler for that DSL subsequently generates GPL code for DSL code and properly blends all GPL code together. True to this approach, on input of a FOCAML program, a FOCAML compiler generates a number of GPL subprograms: one protocol subprogram for every instantiated family signature in the main body (which represents a member of a family of constraint automata) and one main subprogram for the main body itself. Each of these compiler-generated subprograms constitutes a separate syntactic module, where the precise meaning of "module" depends on the GPL (e.g., a package in Java or a collection of functions with a special name prefix in C). Combined with the hand-written worker subprograms in the GPL—also referenced in the main body, through instantiated foreign signatures—these subprograms compose into a full, modular GPL program.

Every worker/protocol/main subprogram defines one or more virtual *units of parallelism* [BST89], henceforth often just called "units": a worker subprogram defines one *worker unit*, a protocol subprogram defines one or more *protocol units*, and the main subprogram defines one *main unit*. The exact number of protocol units defined by a protocol subprogram depends on the specific compilation approach, two basic alternatives of which I explain shortly. At run-time, virtual units of parallelism *map* to physical threads but not necessarily in a one-to-one fashion. As the number of protocol units, this mapping depends on the specific compilation approach.

Figure 4.1 shows two of the most basic compilation approaches. These approaches constitute the two ends of a *spectrum* of compilation approaches that vary in the amount of sequentiality/parallelism in their resulting code: the *Centralized Approach* on the left yields maximally sequential code, whereas the *Distributed Approach* on the right yields maximally parallel code. For those two approaches, and with Java as the complementary GPL, Figures 4.3 and 4.4 exemplify the relation between a specification, the worker/protocol/main subprograms in its implementation, the virtual units of parallelism defined by those subprograms, and the underlying physical threads. Both figures show the same passive objects, which means that the *existence* of those passive objects does not depend on the specific compilation approach. The *content* of those passive objects, however, does differ between the Distributed Approach and the Centralized Approach. In particular, as shown in Figures 4.3 and 4.4, these differences manifest not only in the *number* of protocol units, but they also affect the *mapping* of those units to threads at run-time. I explain these dif-

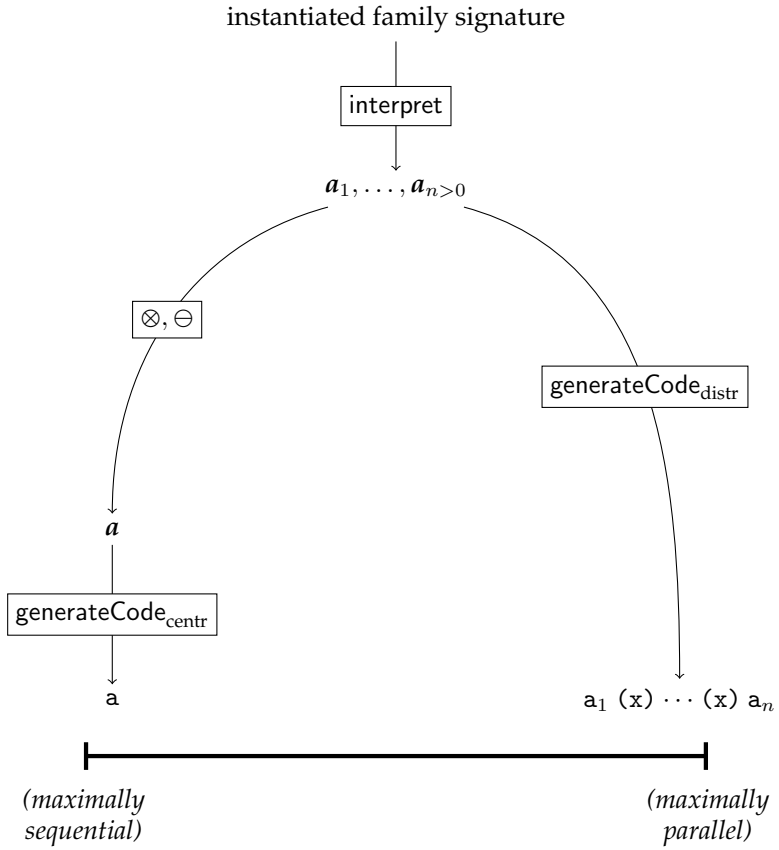


Figure 4.1: Basic compilation approaches

```
1 LossyFifo(in;out) = { LossySync(in;P) mult Fifo(P;out) }

2 main = { LossyFifo(A;B) } among { Producer(A) and Consumer(B) }
```

Figure 4.2: FOCAML definition for family LossyFifo and a main definition for a member of LossyFifo among two workers

ferences in more detail in the next subsections. Regardless of the specific compilation approach, every worker/main unit maps to its own separate thread.

Generally, at run-time, execution of a full GPL program obtained from a FOCAML program starts with the main unit, which (i) constructs port data structures, (ii) constructs and starts protocol units, and (iii) constructs and starts worker units. As part of their construction, the main unit passes port data structures to worker- and protocol units through their parameters. Doing

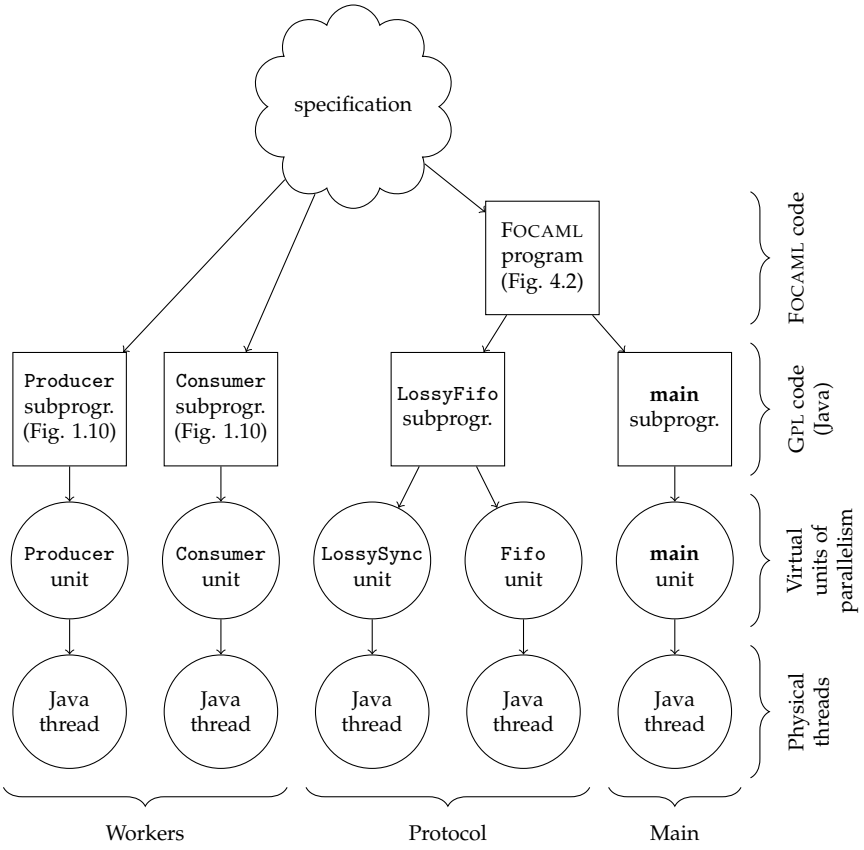


Figure 4.3: Relation between passive objects (squares) and active objects (circles) in the Distributed Approach on input of the FOCAML program in Figure 4.2

so establishes links between those units of parallelism that receive the same port data structure. Afterward, every port data structure has exactly two such units as its "users". Worker units can perform blocking I/O operations on port data structures; protocol units can monitor port data structures for occurrences of such *events*. More precisely, worker units act *proactively*; protocol units act *reactively*, in an event-driven fashion. Whenever a worker unit performs an I/O operation on the data structure for a port p, it informs the "neighboring" protocol unit, which shares access to this data structure, about this *p-event* and afterward becomes suspended until its operation completes. The neighboring protocol unit subsequently resumes from its suspended base state to start a new round of *event-handling*. In every such round, a protocol unit checks if a "suitable" subset of pending I/O operations exists, whose elements would yield an admissible instance of interaction in its current state upon their syn-

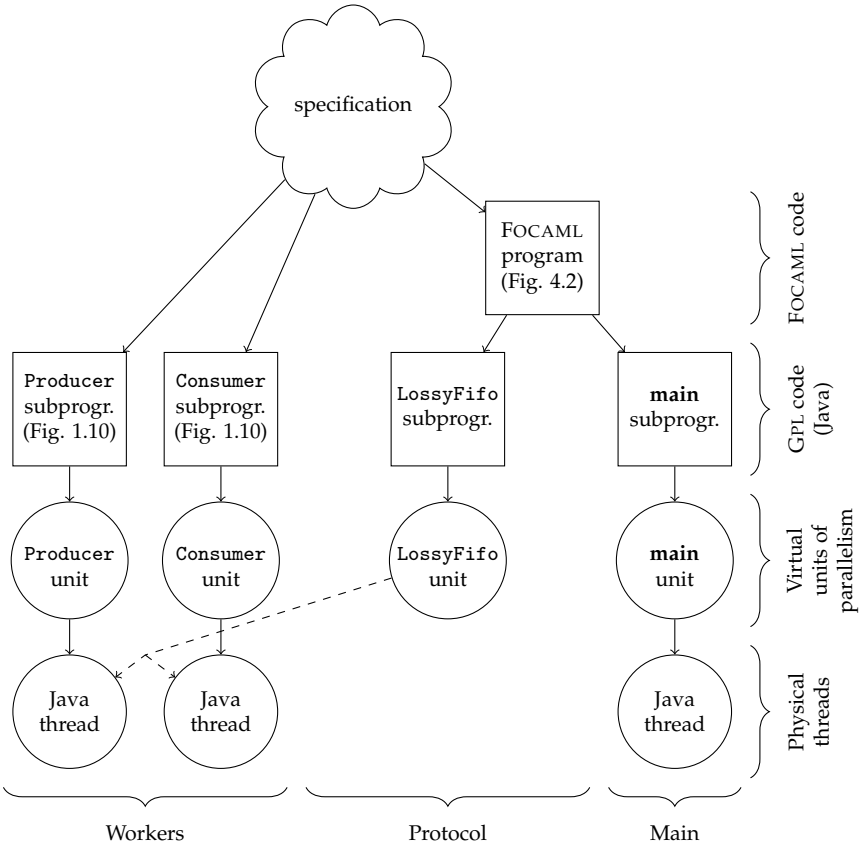


Figure 4.4: Relation between passive objects (squares) and active objects (circles) in the Centralized Approach on input of the FOCAML program in Figure 4.2. Dashed arrows indicate that the `LossyFifo` unit does not map to its own separate thread but "piggybacks" on the threads of the two worker units, as explained in more detail on page 101.

chronous completion. If so, the protocol unit effectuates that instance of interaction: it completes the pending I/O operations involved, thereby allowing the worker units that performed those I/O operations to resume. Otherwise, if no suitable subset exists, (i) the new I/O operation becomes pending on p, (ii) all previously pending I/O operations remain pending, and (iii) the worker unit that performed the I/O operation on p remains suspended.

Distributed Approach

A FOCAML compiler that generates code under the Distributed Approach takes two steps on input of an instantiated family signature. In the first step, the

compiler calls a FOCAML interpreter to obtain a list of the n "small" primitive constraint automata $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ denoted by the input signature. In the second step, the compiler translates this collection of small automata to a GPL protocol subprogram. This protocol subprogram defines n protocol units. Individually, every one of these protocol units locally simulates a small automaton $\boldsymbol{a}_i$; collectively, these protocol units globally simulate the product of $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$. To achieve the latter, the protocol units need to synchronize their local behavior with each other: before doing anything, all protocol units must reach *consensus* about (i) which instances of interaction they admit, given their current local states and pending I/O operations, and (ii) which of those admissible instances they plan on actually effectuating. In other words, these protocol units must reach consensus about (i) which global transitions they can fire by synchronously firing their local transitions and (ii) which of those global transitions they actually plan on firing. Thus, these protocol units effectively multiply $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ at run-time. In Figure 4.1, I denote the code of the n protocol units by $\texttt{a}_1, \ldots, \texttt{a}_n$, the code of their consensus algorithm by `(x)`, and the full protocol subprogram constituted by those pieces of code by $\texttt{a}_1\ \texttt{(x)}\ \cdots\ \texttt{(x)}\ \texttt{a}_n$. Here, the placement of `(x)` between $\texttt{a}_1, \ldots, \texttt{a}_n$ indicates that the consensus algorithm denoted by `(x)` actually multiplies constraint automata (in the sense of Definition 29 of $\otimes$). I consider this a maximally parallel setup, thereby stipulating that the families of constraint automata in the core set in Figure 3.4 have indivisible constraint automata as their members.

As an example, Figure 4.3 shows the relation between passive objects and active objects in the Distributed Approach for the FOCAML program in Figure 4.2. In this case, the protocol subprogram defines two protocol units. Each of these protocol units maps to a separate thread at run-time.

Protocol units in the Distributed Approach need to respond to two kinds of events. *Internal events* occur whenever one protocol unit sends a control message to another protocol unit, through a shared internal port data structure, as part of their consensus algorithm. *Boundary events* occur whenever a worker unit performs an I/O operation on a shared port data structure. (In the Distributed Approach, thus, every port data structure either has a worker unit and a protocol unit as its users or it has two protocol units as its users.) Despite their conceptual differences, a protocol unit can handle internal and boundary events in nearly the same way. Figure 4.5 shows a simplified event-handler for a protocol unit that simulates a small automaton. I do not intend this figure to convey a real "algorithm"; it serves just as a stylized description of what event-handling roughly entails in the Distributed Approach. In particular, for simplicity, Figure 4.5 intentionally misses a number of actually essential steps (e.g., for dealing with cases where a protocol unit simultaneously receives messages from different protocol units, which may cause deadlock if handled uncarefully). Clarke et al. and Proença et al. developed distributed algorithms, run-time systems, and optimization techniques that take these issues into account [CCA07, CP12, CPLA11, PC13a, PC13b, PCdVA11, PCdVA12, Pro11], formulated in terms of mathematical objects structurally different from constraint automata but nevertheless strongly related. I ignore those issues here,

Input: a port p on which an event occurred, a *context* $P^{\text{ctxt}} \subseteq P^{\text{in}} \cup P^{\text{out}}$ of global boundary ports with a pending I/O operation, and the current local state q_i of $\boldsymbol{a}_i$
Output: q_i' holds the next local state of $\boldsymbol{a}_i$.
Effect: either, through the firing of enabled local transitions (including a local transition of $\boldsymbol{a}_i$), an enabled global transition fires (if the I/O operations pending on the ports in P^{ctxt} satisfy that transition's label), or all global transitions are disabled (otherwise).

1. Wake up, and assign q_i to q_i'.
2. Assign $\emptyset$ to Φ, a variable for a set of data constraints.
3. For all transitions $q_i \xrightarrow{P_i, \phi_i}_i q_i'$, ordered nondeterministically:
   (a) If $p \notin P_i$, continue (i.e., skip to the next iteration).
   (b) If $P_i \cap (P^{\text{in}} \cup P^{\text{out}}) \not\subseteq P^{\text{ctxt}}$ (i.e., not all boundary ports involved in the current local transition have a pending I/O operation), continue.
   (c) Assign $\{\phi\}$ to Φ', a variable for a set of data constraints.
   (d) For all ports $p' \in P_i \setminus (P^{\text{in}} \cup P^{\text{out}})$:
      i. Send a message to the protocol unit that shares access to p' to ask which data constraints must hold for that unit to fire a transition involving p'.
      ii. Await an answer message Φ'' from that protocol unit.
      iii. Assign $\{\phi' \wedge \phi'' \mid \phi' \in \Phi' \textbf{ and } \phi'' \in \Phi''\}$ to Φ'.
   (e) Assign $\Phi \cup \Phi'$ to Φ.
4. If the p-event originated from a worker unit:
   (a) For all data constraints $\phi \in \Phi$:
      i. Compute a data assignment σ that respects the pending I/O operations and satisfies ϕ; continue if no such σ exists.
      ii. Distribute data among local ports and memory cells according to σ.
      iii. Send σ to all protocol units sent messages to in Step 3.
      iv. Compute a q' such that $q_i \xrightarrow{P_i, \phi_i} q'$ and $P_i \subseteq \mathrm{Dom}(\sigma)$ and $\sigma \models \phi_i$.
      v. Assign q' to q_i', and abort the loop.
   (b) If the previous loop never made it to Step 4-a-v, send $\emptyset$ (i.e., the empty data assignment) to all protocol units sent messages to in Step 3.

   Else, if the p-event originated from a protocol unit:
   (a) Send an answer message Φ to that protocol unit.
   (b) Await a message with a data assignment σ.
   (c) If $\sigma \neq \emptyset$, distribute data among local ports and memory cells according to σ.
   (d) Send σ to all protocol units sent messages to in Step 3.
   (e) If $\sigma \neq \emptyset$, compute a q' such that $q_i \xrightarrow{P_i, \phi_i} q'$ and $P_i \subseteq \mathrm{Dom}(\sigma)$ and $\sigma \models \phi_i$.
   (f) If $\sigma \neq \emptyset$, assign q' to q_i'.
5. Go dormant.

Figure 4.5: Simplified p-event-handler for a protocol unit that simulates a small automaton $\boldsymbol{a}_i = (Q_i, (P_i^{\text{all}}, P_i^{\text{in}}, P_i^{\text{out}}), M_i, \longrightarrow_i, (q_i^0, \mu_i^0))$ in the Distributed Approach, where P^{in} and P^{out} denote the sets of global input and output ports

because I do not use the Distributed Approach further in this thesis.

Centralized Approach

A FOCAML compiler that generates code under the Centralized Approach takes three steps on input of an instantiated family signature. In the first step, the compiler obtains a list of n small primitive constraint automata $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ as in the Distributed Approach, by calling a FOCAML interpreter. In the second step, the compiler multiplies those small automata and subtracts all internal ports to get one "large" composite constraint automaton $\boldsymbol{a}$. In the third step, the compiler translates this large automaton to a protocol subprogram a in the complementary GPL. This protocol subprogram defines exactly one protocol unit, which simulates $\boldsymbol{a}$. I consider this a maximally sequential setup, because this protocol unit serializes all parallelism among the small automata.

As an example, Figure 4.4 shows the relation between passive objects and active objects in the Centralized Approach for the FOCAML program in Figure 4.2. In contrast to the protocol units in Figure 4.3, the protocol unit in Figure 4.4 does not map to its own separate thread at run-time. Instead, the worker threads execute the interaction code in the compiler-generated protocol subprogram. Doing so results in better performance. To understand this claim, suppose that the protocol unit in Figure 4.4 *does* map to its own separate thread. In that case, every time a worker unit becomes suspended until its I/O operation completes, its thread goes to sleep. Simultaneously, the thread of its neighboring protocol unit awakes in an attempt to handle this new event. The worker unit remains suspended, and its thread remains asleep, at least until the event-handler of its neighboring protocol unit terminates. But then, the sleeping worker thread might as well have executed all event-handling code itself, *on behalf of the neighboring protocol unit*. This would have eliminated some of the overhead in threads going to sleep and waking up. In fact, if all worker threads collectively handle events on behalf of the protocol unit in this way, this protocol unit does not need its own separate thread anymore, eliminating also the general overhead of managing an extra thread. Generally, the smaller the number of threads without sacrificing useful parallelism, the better.

The single protocol unit in the Centralized Approach needs to respond only to boundary events. (In the Centralized Approach, thus, every port data structure has a worker unit and a protocol unit as its users.) Figure 4.5 shows a simplified event-handler for a protocol unit that simulates a large automaton. I do not intend this figure to convey a real "algorithm"; it serves just as a stylized description of what event-handling roughly entails in the Centralized Approach.

Distribution versus Centralization

The Distributed Approach and the Centralized Approach differ essentially in when multiplication takes place. In the Distributed Approach, multiplication occurs on-line, dynamically at run-time; in the Centralized Approach,

Input: a port p on which an event occurred, a context $P^{\text{ctxt}} \subseteq P^{\text{in}} \cup P^{\text{out}}$ of boundary ports with a pending I/O operation, and the current state q
Output: q' holds the next state.
Effect: either an enabled transition fires (if the I/O operations pending on the ports in P^{ctxt} satisfy that transition's label), or all transitions are disabled (otherwise).

1. Wake up, and assign q to q'.
2. For all transitions $q \xrightarrow{P,\phi} q''$, ordered nondeterministically:
   (a) If $p \notin P$, continue.
   (b) If $P \cap (P^{\text{in}} \cup P^{\text{out}}) \not\subseteq P^{\text{ctxt}}$, continue.
   (c) Compute a data assignment σ that respects the pending I/O operations and satisfies data constraint ϕ; continue if no such σ exists.
   (d) Distribute data among ports and memory cells according to σ.
   (e) Assign q'' to q'.
   (f) Abort the loop.
3. Go dormant.

Figure 4.6: Simplified p-event-handler for a protocol unit that simulates a large automaton $(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0))$ in the Centralized Approach

```
1  Alice(in;out) = { Sync(in;out) }
2  Bob  (in;out) = { Sync(in;out) }
3  Carol(in;out) = { Sync(in;out) }
4  Dave (in;out) = { Sync(in;out) }

5  AliceBobCarol(in;out) = {
6    Alice(in,P1) mult Bob(P1,P2) mult Carol(P2;out)
7  }

8  AliceBobCarolDave(in[];out[]) = {
9    AliceBobCarol(in[1];out[1]) mult Dave(in[2];out[2])
10 }

11 main = {
12     AliceBobCarolDave(A[1..2];B[1..2])
13   } among {
14     Producer(A[1]) and Producer(A[2]) and Consumer(B[1]) and Consumer(B[2])
15   }
```

Figure 4.7: Alice, Bob, Carol, and Dave in FOCAML

multiplication occurs off-line, statically at compile-time. Consequently, in the Distributed Approach, compilation requires few resources while execution requires many (i.e., the consensus algorithm), while in the Centralized Approach, compilation requires many resources (i.e., the product computation) while execution requires few.

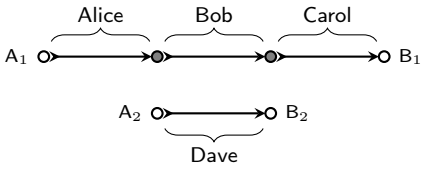


Figure 4.8: Alice, Bob, Carol, and Dave in Reo

I define the *latency* of a compiler-generated protocol subprogram as the average number of time units required to fire a transition (i.e., effectuate one instance of interaction); I define its *throughput* as the average number of transitions fired per time unit. The Distributed Approach and the Centralized Approach have opposite latency/throughput characteristics. To illustrate this point, suppose that I compile the FOCAML program in Figure 4.7 twice (see also its Reo equivalent in Figure 4.8), once for every approach.

- The protocol subprogram generated under the Distributed Approach defines four protocol units, which I anthropomorphize as Alice, Bob, Carol, and Dave. Each of these protocol units maps to its own thread. Whenever an I/O operation occurs on `A[1]`, Alice awakes to handle this event (see Figure 4.5). She then asks Bob which data constraints must hold for him to fire a transition involving the port data structure shared between Alice and Bob. Bob awakes upon receiving Alice's message and, in turn, asks Carol which data constraints must hold for her to fire a transition involving the port data structure shared between Bob and Carol. Upon receiving Bob's message, also Carol awakes. If `B[1]` has a pending I/O operation, she replies the data constraint on her transition to Bob; otherwise, she replies $\perp$. Once Bob has received Carol's reply, he prepares and replies a set of data constraints to Alice based on the message he received from Carol. Alice can subsequently determine whether she, Bob, and Carol can synchronously fire local transitions that compose into an admissible global transition, by searching for a satisfying data assignment for one of the compound data constraints that she received from Bob. Note that, to find such a data assignment, Alice does not need access to port data structures other than her own (i.e., `A[1]`, whose pending `put` contains the datum to propagate to Carol via Bob). Once computed, Alice sends the satisfying data assignment to Bob, after which Bob sends this assignment to Carol. Carol, finally, looks up the datum assigned to the variable for `B[1]` in this satisfying data assignment and exchanges this datum through `B[1]` to the worker that performed a `get` on `B[1]` (i.e., as Alice, also Carol needs access only to her own port data structures). In parallel to the communication between Alice, Bob, and Carol, by Definition 29 of $\otimes$ (which admits true concurrency), Dave can independently try to fire his transition whenever an I/O operation occurs on `A[2]` or `B[2]`. After all, Dave controls the interaction on port data structures that

Alice, Bob, and Carol do not know about and vice versa.

Thus, the distributed nature of Alice, Bob, and Carol (i.e., the communication necessary for them to synchronously fire their local transitions) negatively affects latency; the parallelism between them and Dave (i.e., the ability of Dave to fire his transition independently of theirs) positively affects throughput.

- The protocol subprogram generated under the Centralized Approach defines only one protocol unit. Whenever an I/O operation occurs on `A[1]`, this protocol unit checks if its transition involving `A[1]` (i.e., the transition composed of Alice's, Bob's, and Carol's local transitions) can fire. While this happens, however, the protocol unit cannot fire other transitions. In other words, if an I/O operation occurs on `A[2]` just after the occurrence of the I/O operation on `A[1]`, the protocol unit cannot try to fire its transition involving `A[2]` (i.e., Dave's transition) as long as it has not finished handling the event on `A[1]`.

  Thus, the centralized nature of the single protocol unit for Alice, Bob, Carol, and Dave (i.e., the fact that it does not require a consensus algorithm and, as such, avoids a major source of overhead) positively affects latency; its sequentiality (i.e., its inability to simultaneously fire multiple transitions) negatively affects throughput.

In summary, a FOCAML compiler that generates code under the Distributed Approach generally yields protocol subprograms with high latency and high throughput; a FOCAML compiler that generates code under the Centralized Approach generally yields protocol subprograms with low latency and low throughput. The compiler that I present in Section 4.2 generates code under the Centralized Approach. In Chapter 5, I refine this approach to recover "useful parallelism", thereby improving throughput.

4.2 Practice

(I have not yet submitted the material in this section for publication.)

Compiler

I extended the basic Eclipse editor/parser/interpreter plug-in for FOCAML, presented in Chapter 3, with a FOCAML compiler that generates Java code. This FOCAML-to-Java compiler, called Lykos , closely follows the Centralized Approach as presented in Section 4.1. To multiply small automata (resulting from interpreting an instantiated family signature) into a large automaton, as the Centralized Approach demands, Lykos uses the Java library for representing constraint automata and their operations in Chapter 2. To generate code for a large automaton, then, Lykos uses ANTLR's *StringTemplate* technology. StringTemplate consists of a grammar for writing *templates* and a Java library

through which to invoke a *template engine*. Given a template and a set of data, the template engine "fills" the "holes" in the template with that data. In the case of Lykos, the template consists of Java code and the set of data consists of a data structure for the large automaton computed previously.

This template approach to generating code easily supports new target languages: the template-to-fill forms the only truly Java-specific aspect of Lykos (along with the Java run-time library to actually run the generated code, of course). This means that extending Lykos to other target languages requires relatively little effort: just write a new template. As concrete evidence in support of this claim, a master student has recently extended Lykos with a template to generate C code, only as a *minor* part of his MSc thesis. For the run-time library, a target language should support just some form of threading and mutual exclusion (i.e., I do not use any exotic Java-specific features), which most—if not all—modern GPLs do.

The output of Lykos consists of a program-independent run-time library, a custom main subprogram, a custom protocol subprogram, and a number of `Runnable` classes that wrap around hand-written worker subprograms. Each of these subprograms defines one virtual unit of parallelism. Every worker unit maps to its own thread. The resulting worker threads execute not only computation code but also interaction code, on behalf of the protocol unit defined by the protocol subprogram (as explained in Section 4.1). Thus, the protocol unit does not map to its own separate thread at run-time.

Run-Time Library

Figure 4.9 shows the part of the run-time library concerned with *contexts*.

- Every `Context` represents a registry of the I/O operations pending on the input and output ports of a constraint automaton (where "input" and "output" qualify ports from the protocol perspective).

  A `Context` has a field for storing an array of `AtomicIntegers` to enable lock-free concurrent accesses and updates with bitwise operators. Each of the 32 bits in an `AtomicInteger` represents the (un)availability of an I/O operation on a port. For instance, suppose that I have ports A, B, and C. Indexing from right to left, the following integers—in Java 7 syntax—denote sets {A}, {A, C}, and {A, B, C}:

  ```
  0b0000_0000_0000_0000_0000_0000_0000_0001
  0b0000_0000_0000_0000_0000_0000_0000_0101
  0b0000_0000_0000_0000_0000_0000_0000_0111
  ```

  To (un)register ports in a `Context`, a thread should first compute their corresponding integer `mask`. Subsequently, it can `add` or `remove` that `mask`, using Java's bitwise operators to efficiently manipulate the `Context`'s internal `integers`. Similarly, threads use bitwise operators to check if a `Context` `contains` certain ports.

```
1  public class Context {
2    public final AtomicInteger[] integers;
3
4    public Context(int nPorts) {
5      this.integers = new AtomicInteger[(nPorts / 32) + 1];
6      for (int i = 0; i < this.integers.length; i++)
7        this.integers[i] = new AtomicInteger();
8    }
9
10   public void add(int index, int mask) {
11     AtomicInteger integer = integers[index];
12     int bits = integer.get();
13     while (!integer.compareAndSet(bits, bits | mask))
14       bits = integer.get();
15   }
16
17   public boolean contains(int index, int mask) {
18     return mask == (integers[index].get() & mask);
19   }
20
21   public void remove(int index, int mask) {
22     AtomicInteger integer = integers[index];
23     int current = integer.get();
24     while (!integer.compareAndSet(current, current & ~mask))
25       current = integer.get();
26 } }
```

Figure 4.9: Java run-time library (contexts)

Figure 4.10 shows the part of the run-time library concerned with automata, states, and transitions.

- Every `Automaton` represents a constraint automaton. An `Automaton` has fields for storing a `Context` to register pending I/O operations and a `Semaphore` (from package `java.util.concurrent`) to guarantee mutual exclusion among threads trying to execute code on its behalf (i.e., worker threads). Compiler-generated subclasses of `Automaton` typically have a number of extra fields for storing `State`s and bookkeeping information (e.g., to account for the initial/current state). In the next subsection, I give examples of compiler-generated subclasses of `Automaton`.

- Every `State` represents a state in a constraint automaton. Implementations of `State` typically have a number of fields for storing outgoing `Transition`s and bookkeeping information (e.g., to account for fairness). Threads can cause an `Automaton` to `reach` a `State`, making that `State` the current `State` of that `Automaton`. In the next subsection, I give examples of compiler-generated implementations of `State`.

- Every `Current` represents the current state in a constraint automaton.

- Every `Transition` represents a transition out of a state in a constraint automaton. In method `checkDataConstraint`, the current thread checks

```
1  public abstract class Automaton extends Thread {
2    public final Context context;
3    public final Semaphore semaphore = new Semaphore(1);
4
5    public Automaton(int nPorts) {
6      this.context = new Context(nPorts);
7  } }

8  public interface State {
9    public void reach();
10 }

11 public class Current {
12   public volatile State state;
13 }

14 public abstract class Transition {
15   protected boolean checkDataConstraint() {
16     return true;
17   }
18
19   protected abstract boolean fire();
20 }
```

Figure 4.10: Java run-time library (automata, states, transitions)

whether the data constraint of the `Transition` involved holds true given the currently pending I/O operations and the content of memory cells. The default implementation simply returns **true**; usually, compiler-generated subclasses of `Transition` override this implementation. Threads can attempt to make a transition by invoking method `fire`. If successful, this method returns **true**; otherwise, it returns **false**. In the next subsection, I give examples of compiler-generated subclasses of `Transition`.

Figure 4.11 shows the part of the run-time library concerned with event-handlers.

- Every `Handler` represents an event-handler for I/O operations on a particular port. A `Handler` has fields for storing a `Semaphore`, passed to it through its constructor, to guarantee mutual exclusion in executing code of its corresponding `Automaton` (i.e., the `Semaphore` passed to a `Handler` and the `Semaphore` on line 3 in Figure 4.10 should refer to the same object).

  Whenever a worker thread performs an I/O operation, it should first `register` that I/O operation in a `Context` via the appropriate `Handler`. Afterward, until *some* thread has completed the I/O operation, worker threads can use method `callSync` to execute the actual event-handling code in method `call`, under mutual exclusion (i.e., only one `Handler` of an `Automaton` may run at a time). If this method returns **false**, the worker

```
21 public abstract class Handler {
22   public final Semaphore semaphore;
23
24   public Handler(Semaphore semaphore) {
25     this.semaphore = semaphore;
26   }
27
28   public boolean callSync() throws InterruptedException {
29     semaphore.acquire();
30     boolean isCompleted = call();
31     semaphore.release();
32     return isCompleted;
33   }
34
35   public abstract boolean call();
36   public abstract void register();
37 }
```

Figure 4.11: Java run-time library (event-handlers)

```
1  public enum IO { PERFORMED, COMPLETED }

2  public abstract class Port {
3    public final Semaphore semaphore = new Semaphore(0);
4
5    public volatile Handler handler;
6    public volatile IO status;
7    public volatile Object buffer;

8  public class MemoryCell {
9    public volatile Object content;
10 }
```

Figure 4.12: Java run-time library (ports and memory cells)

thread has failed to complete the I/O operation. In that case, the I/O operation should remain pending. In the next subsection, I give examples of compiler-generated implementations of `Handler`.

Figure 4.12 shows the part of the run-time library concerned with ports and memory cells.

- Every `Port` represents a port controlled by a constraint automaton.

  A `Port` has a field `buffer` to store the datum involved in a pending I/O operation, a field `status` to store the status of a pending I/O operation, a `Handler` for events caused by `puts` or `gets` (depending on the direction of the `Port`), and a `Semaphore` for threads to block on (until an I/O operation becomes `COMPLETED`).

- Every `MemoryCell` represents a memory cell in a constraint automaton.

```
1  public class OutputPortImpl extends Port implements OutputPort {
2    public void put(Object datum) throws InterruptedException {
3      buffer = datum;
4      status = IO.PENDING;
5      handler.register();
6      resume();
7    }
8
9    public void putUninterruptibly(Object datum) {
10     while (true)
11       try {
12         put(datum);
13         return;
14       } catch (InterruptedException exc) { break; }
15     while (true)
16       try {
17         resume();
18         return;
19       } catch (InterruptedException exc) {}
20   }
21
22   public void resume() throws InterruptedException {
23     while (status != IO.COMPLETED && !handler.callSync())
24       semaphore.acquire();
25 } }

26 public class InputPortImpl extends Port implements InputPort {
27   public Object get() throws InterruptedException {
28     buffer = null;
29     status = IO.PENDING;
30     handler.register();
31     return resume();
32   }
33
34   public Object getUninterruptibly() {
35     while (true)
36       try {
37         return get(datum);
38       } catch (InterruptedException exc) { break; }
39     while (true)
40       try {
41         return resume();
42       } catch (InterruptedException exc) {}
43   }
44
45   public Object resume() throws InterruptedException {
46     while (status != IO.COMPLETED && !handler.callSync())
47       semaphore.acquire();
48     return buffer;
49 } }
```

Figure 4.13: Implementation of the Java API for ports in Figure 1.9

Finally, Figure 4.13 shows the implementation of the Java API for ports in Figure 1.9, based on class `Port` in Figure 4.12. (These figures constitute one of the rare exceptions in this thesis, where I qualify ports as "input" or "output" from the perspective of workers.) Whenever a worker thread `puts` a `datum` to an `OutputPortImpl`, (i) it temporarily stores that `datum` in field `buffer`, (ii) it updates field `status` to remember that it has `PERFORMED` an I/O operation, (iv) it `registers` the I/O operation through the `Handler`, and (v) it actually runs the `Handler` through method `callSync`. If method `callSync` returns **true**, in which case the worker thread has `COMPLETED` the `put`, the worker thread immediately returns; otherwise, the worker thread blocks on a `Semaphore` either until it gets a new chance to complete the `put` itself or until another thread has done so on its behalf. Should an `InterruptedException` occur during a `put`, the interrupted thread can later resume this I/O operation using method `resume`. Method `putUninterruptibly` demonstrates this idiom. Methods `get` and `resume` of `OutputPortImpl` work similarly. Software engineers should write their programs against interfaces `OutputPort` and `InputPort`; they should never use classes `OutputPortImpl` and `InputPortImpl` directly.

I concentrated on the core functionality of the run-time library, intentionally omitting some of its more advanced features such as I/O operations with timeouts and internal transitions. Although Lykos supports these features, their explanation goes beyond my current intent of giving only a broad overview.

Compiler-Generated Code

To exemplify compiler-generated code, Figure 4.15 and further show the code generated by Lykos on input of instantiated family signature `LateAsyncMerger2(A,B;C)` (cf. Figure 3.3), with compiler flag `IGNORE_DATA` raised. When raised, this flag signals to Lykos that data do not matter. Lykos subsequently generates code in which transitions can fire without checking their data constraints. Abstracting away data constraints in this way makes it easier to explain and understand the general structure of the generated code. In Chapter 7, I discuss checking data constraints in more detail. In practice, software engineers may raise the `IGNORE_DATA`-flag whenever they want to use a data-aware protocol in a data-unaware fashion (i.e., as a pure data-insensitive synchronization protocol).

First, Figure 4.14 shows a typical `main` method that uses a `Protocol` (discussed shortly) to control the interaction between two `Producers` and a `Consumer` in Figure 1.10. (This figure constitutes one of the rare exceptions in this thesis, where I qualify ports as "input" or "output" from the perspective of workers.) As this figure shows, sharing ports (i.e., passing the same `Port` to multiple constructors of `Protocol/Producer/Consumer`) links workers to protocols. Lykos automatically generates a `main` method similar to the one in Figure 4.14, thereby producing a full executable program.

Figure 4.15 shows class `Protocol`. Instances of this class fully encapsulate the $\mathsf{LateAsyncMerger}_2$ protocol, first informally described on page 7 and later formalized as a constraint automaton in Figure 2.2. A `Protocol` has fields

```
1  public class ProducersConsumerProgram {
2    public static void main(String[] args) {
3      OutputPort A = Ports.newOutputPort();
4      OutputPort B = Ports.newOutputPort();
5      InputPort C = Ports.newInputPort();
6      new Protocol((Port) A, (Port) B, (Port) C);
7      (new Producer(A)).start();
8      (new Producer(B)).start();
9      (new Consumer(C)).start();
10 } }
```

Figure 4.14: Producers/consumer program for LateAsyncMerger$_2$ in Java, generated for `LateAsyncMerger2(A,B;C)`

```
1  public class Protocol {
2    final Automaton7 automaton7;
3    final Port A;
4    final Port B;
5    final Port C;
6    final MemoryCell memoryCell1 = new MemoryCell();
7
8    public Protocol(Port A, Port B, Port C) {
9      this.A = A;
10     this.B = B;
11     this.C = C;
12     this.automaton7 = new Automaton7();
13     initialize();
14   }
15
16   public void initialize() {
17     this.A.handler = new HandlerForA(this);
18     this.B.handler = new HandlerForB(this);
19     this.C.handler = new HandlerForC(this);
20     this.automaton7.initialize(this);
21 } }
```

Figure 4.15: Class `Protocol`, generated for `LateAsyncMerger2(A,B;C)`

for storing an `Automaton` (which represents the constraint automaton in Figure 2.2), three `Ports` (which represent its input ports A and B and its output port C, where "input" and "output" qualify ports from the protocol perspective), and a `MemoryCell` (which represents its memory cell x). In the constructor, the current thread stores the provided `Port` arguments in their corresponding fields. Subsequently, the current thread creates and stores a new `Automaton`, namely an instance of compiler-generated subclass `Automaton7` (discussed shortly). In method `initialize`, the current thread creates and stores new `Handlers` in the appropriate fields of `A`, `B`, and `C`, namely instances of compiler-generated classes `HandlerForA`, `HandlerForB`, and `HandlerForC` (discussed shortly). Subsequently, the current thread further initializes the previously constructed `Automaton`.

```
1  class Automaton7 extends Automaton {
2    final Automaton7State1 state1;
3    final Automaton7State2 state2;
4    final Current current = new Current();
5
6    public Automaton7() {
7      super(3);
8      this.state1 = new Automaton7State1();
9      this.state2 = new Automaton7State2();
10   }
11
12   public void initialize(Protocol protocol) {
13     this.state1.initialize(protocol);
14     this.state2.initialize(protocol);
15     this.state1.reach();
16 } }
```

Figure 4.16: Class `Automaton7`, generated for `LateAsyncMerger(A,B;C)`

Figure 4.16 shows class `Automaton7`. (The "7" has no real significance; it serves just as an internal identifier during compilation.) Every instance of this class represents the constraint automaton in Figure 2.2. An `Automaton7` has fields for storing two `States` (which represent the states in the constraint automaton in Figure 2.2) and a `Current` (which represents its current state). In the constructor, the current thread creates and stores two new `States`, namely instances of compiler-generated subclasses `Automaton7State1` and `Automaton7State2` (discussed shortly). In method `initialize`, the current thread further initializes the previously constructed `States`. Subsequently, the current thread sets `state1` as the initial/current `State`, by invoking method `reach`.

Figure 4.17 shows classes `Automaton7State1` and `Automaton7State2`. Every instance of the former class represents the left state in Figure 2.2; every instance of the latter class represents the right state. An `Automaton7State1` has fields for storing two `Transitions` (which represent the outgoing transitions from the left state in Figure 2.2), a `Current` (which represents the current state of the constraint automaton), and two `Ports` (which represent ports A and B, involved in those transitions). In the constructor, the current thread creates and stores two new `Transitions`, namely instances of compiler-generated subclasses `Automaton7Transition1` and `Automaton7Transition2` (discussed shortly). In method `initialize`, the current thread sets the remaining uninitialized fields with information from the provided `Protocol` argument. Subsequently, the current thread further initializes the previously constructed `Transitions`. In method `reach`, the current thread updates the `current.state` and `releases` a permit to `A`'s and `B`'s `semaphore`. These new permits may wake up worker threads (who previously invoked method `acquire` on line 24 in Figure 4.12), thereby offering them another attempt at handling their still-pending I/O operations (by invoking method `callSync` on line 23 in Figure 4.12). Such another attempt must take place, because following a state change, a new set of admissible transitions becomes available, which affects the potential for I/O

```
1  class Automaton7State1 implements State {
2    final Automaton7Transition1 transition1;
3    final Automaton7Transition2 transition2;
4
5    Current current;
6    Port A;
7    Port B;
8
9    public Automaton7State1() {
10     this.transition1 = new Automaton7Transition1();
11     this.transition2 = new Automaton7Transition2();
12   }
13
14   public void initialize(Protocol protocol) {
15     this.current = protocol.automaton7.current;
16     this.A = protocol.A;
17     this.B = protocol.B;
18     this.transition1.initialize(protocol);
19     this.transition2.initialize(protocol);
20   }
21
22   @Override
23   public void reach() {
24     current.state = this;
25     A.semaphore.release();
26     B.semaphore.release();
27 } }
28 class Automaton7State2 implements State {
29   final Automaton7Transition3 transition3;
30
31   Current current;
32   Port C;
33
34   public Automaton7State2() {
35     this.transition3 = new Automaton7Transition3();
36   }
37
38   public void initialize(Protocol protocol) {
39     this.current = protocol.automaton7.current;
40     this.C = protocol.C;
41     this.transition3.initialize(protocol);
42   }
43
44   @Override
45   public void reach() {
46     current.state = this;
47     C.semaphore.release();
48 } }
```

Figure 4.17: Java code generated for `LateAsyncMerger(A,B;C)`—class `Automaton7State1`

```
1  class Automaton7Transition1 extends Transition {
2    Context context;
3    Port A;
4    Automaton7State2 target;
5
6    public initialize(Protocol protocol) {
7      this.context = protocol.automaton7.context;
8      this.A = protocol.A;
9      this.target = protocol.automaton7.state2;
10   }
11
12   protected boolean checkSynchronizationConstraint() {
13     return true && context.contains(0, 0b00000000000000000000000000000001);
14   }
15
16   @Override
17   protected boolean fire() {
18     boolean canFire = checkSynchronizationConstraint() && checkDataConstraint();
19     if (canFire) {
20       context.remove(0, 0b00000000000000000000000000000001);
21       A.status = IO.COMPLETED;
22       A.semaphore.release();
23       target.reach();
24     }
25     return canFire;
26 } }
```

Figure 4.18: Class `Automaton7Transition1`, generated for `LateAsyncMerger(A,B;C)`

operations to complete. For instance, the right state in Figure 2.2 has no outgoing transitions involving port A. So, if a worker thread invokes `A.put`, while the `Automaton7` has `state2` as its `current.state` (which represents the right state in Figure 2.2), method `callSync` invoked on line 23 in Figure 4.12 returns **false** after which the worker thread goes to sleep on the next line. As soon as the `Automaton7` reaches `state1`, however, the worker thread should wake up to invoke method `callSync` again. After all, the left state in Figure 2.2 (represented by `state1`) has an outgoing transition involving port A, so the pending I/O operation on A can now complete.

Figure 4.18 shows class `Automaton7Transition1`. Every instance of this class represents the {A}-transition in Figure 2.2, from the left state to the right state. An `Automaton7Transition1` has fields for storing a `Context`, a `Port` (which represents port A involved in the {A}-transition in Figure 2.2), and a `State` (which represents its target state). In method `initialize`, the current thread sets these fields with information from the provided `Protocol` argument. In method `checkSynchronizationConstraint`, the current thread checks if the `context` has a bit set for `A`. In method `fire`, the current thread first checks the synchronization constraint and the data constraint. If those constraints hold true, the current thread unsets the bit previously set for `A` in the `context`, it updates the `A.status` accordingly, and it wakes up the worker

thread that performed an I/O operation on A. Subsequently, the current thread sets target as the current.state, by invoking method reach.

Because I raised the IGNORE_DATA-flag, Automaton7Transition1 just inherits its method checkDataConstraint from superclass Transition, whose trivial implementation simply returns **true** (see Figure 4.10), thereby effectively ignoring data constraints. Without this flag raised, to properly deal with data constraints, Lykos generates code that calls a simple *constraint solver* with *forward checking* [Apt09a, BMFL02]. Essentially, for a given data constraint ϕ, this constraint solver tries to find a data assignment σ such that $\sigma \models \phi$ (i.e., σ satisfies ϕ). Because the constraint solver that I use does not advance the state-of-the-art in constraint solving, I skip a further explanation of its workings for now; I discuss data constraints in more detail in Chapters 6 and 7.

Figure 4.19 shows classes Automaton7Transition2 and Automaton7Transition3. Every instance of the former class represents the {B}-transition in Figure 2.2, from the left state to the right state; every instance of the latter class represents the {C}-transition, from the right state to the left state. To highlight their differences, I grayed out the similar parts in Figure 4.19 with respect to Figure 4.18.

Figure 4.20 shows class HandlerForA. Every instance of this class represents an event-handler for I/O operations on port A. An HandlerForA has fields for storing a Context, a Port (which represents port A in the constraint automaton in Figure 2.2), a Current (which represents its current state), and a State (which represents its left state). In the constructor, the current thread sets these fields with information from the provided Protocol argument. In method call, the current thread first checks if the previously performed I/O operation has already completed. If so, the current thread returns. Otherwise, the current thread checks if current.state equals state1, and if so, whether the Transition out of this state can fire. If so, the current thread returns. Otherwise, the current thread removes all permits from A's semaphore (to avoid excessive awakenings) and returns.

Figure 4.21 shows classes HandlerForB and Automaton7HandlerForC. Every instance of the former class represents an event-handler for I/O operations on port B; every instance of the latter class represents an event-handler for I/O operations on port C. To highlight their differences, I grayed out the similar parts in Figure 4.21 with respect to Figure 4.20.

The Handlers in Figures 4.20 and 4.21 handle I/O operations in only one State by attempting to fire only one Transition. This makes these Handlers rather simple. Generally, however, Handlers may handle I/O operations in any number of states by attempting to fire any number of Transitions. Figure 4.22 shows class HandlerForABC to exemplify the general pattern in such cases. Every instance of this Handler represents a comprehensive event-handler for Automaton7 (i.e., not tied to any particular Port), which any thread can call at any time in an attempt to fire any Transition. Normally, Lykos does not generate such comprehensive Handlers, because per-port Handlers, which attempt to fire only those Transitions that actually involve the Port on which an I/O operation became pending, have lower overhead. After all,

```
class Automaton7Transition2 extends Transition {
  Context context;
  Port B;
  Automaton7State2 target;

  public initialize(Protocol protocol) {
    this.context = protocol.automaton7.context;
    this.B = protocol.B;
    this.target = protocol.automaton7.state2;
  }

  protected boolean checkSynchronizationConstraint() {
    return true && context.contains(0, 0b00000000000000000000000000000010);
  }

  @Override
  protected boolean fire() {
    boolean canFire = checkSynchronizationConstraint() && checkDataConstraint();
    if (canFire) {
      context.remove(0, 0b00000000000000000000000000000010);
      B.status = IO.COMPLETED;
      B.semaphore.release();
      target.reach();
    }
    return canFire;
} }

class Automaton7Transition3 extends Transition {
  Context context;
  Port C;
  Automaton7State2 target;

  public initialize(Protocol protocol) {
    this.context = protocol.automaton7.context;
    this.C = protocol.C;
    this.target = protocol.automaton7.state1;
  }

  protected boolean checkSynchronizationConstraint() {
    return true && context.contains(0, 0b00000000000000000000000000000100);
  }

  @Override
  protected boolean fire() {
    boolean canFire = checkSynchronizationConstraint() && checkDataConstraint();
    if (canFire) {
      context.remove(0, 0b00000000000000000000000000000100);
      C.status = IO.COMPLETED;
      C.semaphore.release();
      target.reach();
    }
    return canFire;
} }
```

Figure 4.19: Classes `Automaton7Transition2` and `Automaton7Transition3`, generated for `LateAsyncMerger(A,B;C)`

```
1  class HandlerForA extends Handler {
2    final Context context;
3    final Port A;
4    final Current current;
5    final Automaton7State1 state1;
6
7    public HandlerForA(Protocol protocol) {
8      super(protocol.automaton7.semaphore);
9      this.context = protocol.automaton7.context;
10     this.A = protocol.A;
11     this.current = protocol.automaton7.current;
12     this.state1 = protocol.automaton7.state1;
13   }
14
15   @Override
16   public boolean call() {
17     if (A.status == IO.COMPLETED) return true;
18     if (current.state == state1 && state1.transition1.fire()) return true;
19     A.semaphore.drainPermits();
20     return false;
21   }
22
23   @Override
24   public void register() {
25     context.add(0, 0b00000000000000000000000000000001);
26 } }
```

Figure 4.20: Class `HandlerForA`, generated for `LateAsyncMerger(A,B;C)`

performing an I/O operation on a `Port` can *never* cause method `fire` to return **true** on a `Transition` that does not even involve that `Port`; making such superfluous `fire` invocations only degrades performance. So, I wrote the code in Figure 4.22 by hand and show it here only to exemplify the general code structure of `Handlers`. `HandlerForABC` in Figure 4.22 differs from the previous `Handlers` in Figures 4.20 and 4.21 primarily in the **for**-loop on lines 23–27. In this loop, the current thread iterates over the outgoing `Transitions` of `Automaton7State1` until it successfully `fires` one. To guarantee a limited form of fairness, threads use `trFromState1Index` to remember the index of the `Transition` that most recently successfully `fired` in `state1`. Then, the next time a thread executes the **for**-loop, it will attempt to `fire` that `Transition` only *last*, thereby giving priority to the other outgoing `Transitions`.

API for Ports

Typically, when using FOCAML, software engineers see neither the run-time library nor the compiler-generated code that I presented in this section up to now—at least they do not have to. Instead, software engineers see only the API for ports, which I presented already in Figure 1.9, in Chapter 1 (although for simplicity, I omitted variants of `put` and `get` with timeouts from that figure). Two basic examples of the usage of this API in worker subpro-

```
1  class HandlerForB extends Handler {
2    final Context context;
3    final Port B;
4    final Current current;
5    final Automaton7State1 state1;
6
7    public HandlerForB(Protocol protocol) {
8      super(protocol.automaton7.semaphore);
9      this.context = protocol.automaton7.context;
10     this.B = protocol.B;
11     this.current = protocol.automaton7.current;
12     this.state1 = protocol.automaton7.state1;
13   }
14
15   @Override
16   public boolean call() {
17     if (B.status == IO.COMPLETED) return true;
18     if (current.state == state1 && state1.transition2.fire()) return true;
19     B.semaphore.drainPermits();
20     return false;
21   }
22
23   @Override
24   public void register() {
25     context.add(0, 0b00000000000000000000000000000010);
26 } }
27 class HandlerForC extends Handler {
28   final Context context;
29   final Port C;
30   final Current current;
31   final Automaton7State2 state2;
32
33   public HandlerForC(Protocol protocol) {
34     super(protocol.automaton7.semaphore);
35     this.context = protocol.automaton7.context;
36     this.C = protocol.C;
37     this.current = protocol.automaton7.current;
38     this.state2 = protocol.automaton7.state2;
39   }
40
41   @Override
42   public boolean call() {
43     if (C.status == IO.COMPLETED) return true;
44     if (current.state == state2 && state2.transition3.fire()) return true;
45     C.semaphore.drainPermits();
46     return false;
47   }
48
49   @Override
50   public void register() {
51     context.add(0, 0b00000000000000000000000000000100);
52 } }
```

Figure 4.21: Classes `HandlerForB` and `HandlerForC`, generated for `LateAsyncMerger(A,B;C)`

```
class HandlerForABC extends Handler {
  final Context context;
  final Current current;
  final Automaton7State1 state1;
  final Automaton7State1 state2;
  final Transition[] trFromState1;

  int trFromState1Index = 0;

  public HandlerForABC(Protocol protocol) {
    super(protocol.automaton7.semaphore);
    this.context = protocol.automaton7.context;
    this.current = protocol.automaton7.current;
    this.state1 = protocol.automaton7.state1;
    this.state2 = protocol.automaton7.state2;
    this.trFromState1 = new Transition[] {
      this.state1.transition1, this.state1.transition2
  };}

  @Override
  public boolean call() {
    if (current.state == state1)
      for (int i = trFromState1Index; i < trFromState1Index + 2; i++)
        if (trFromState1[i % 2].fire()) {
          trFromState1Index = (i + 1) % 2;
          return true;
        }
    if (current.state == state2 && state2.transition3.fire()) return true;
    return false;
  }

  @Override
  public void register() {
    throw new UnsupportedOperationException();
} }
```

Figure 4.22: Class `HandlerForABC`, hand-written for `LateAsyncMerger(A,B;C)`

```
public class Benchmark {
  public static AtomicLong N_GETS;
  public static AtomicLong N_PUTS;
  public static CyclicBarrier BARRIER;
  public static Semaphore SEMAPHORE;
}
```

Figure 4.23: Java code for the performance experiments in this thesis (I)

grams (`Producer` and `Consumer`) appeared already in Figure 1.10, in Chapter 1. Another four examples of the usage of this API in worker subprograms (`PortBasedMaster`, `PortBasedSlave`, `PortBasedRelayRaceMaster`, and `PortBasedRelayRaceSlave`) appeared already in Figure 3.33, in Chapter 3.

As a further illustration, Figures 4.23–4.25 show two more example worker

```
1  public class BenchmarkProducer extends Thread {
2    private Datum datum;
3    private OutputPort port;
4
5    public BenchmarkProducer(OutputPort port) {
6      this.port = port;
7    }
8
9    @Override
10   public void run() {
11     warmUp();
12     try {Benchmark.BARRIER.await();} catch (Exception exc) {System.exit(1);}
13     measure();
14   }
15
16   private void warmUp() {
17     Benchmark.SEMAPHORE.release();
18     try {
19       while (!Thread.interrupted()) port.put(0);
20     } catch (InterruptedException exception) {}
21   }
22
23   private void measure() {
24     int i = 0;
25     try {
26       port.resume();
27       while (!Thread.interrupted()) {
28         port.put(0);
29         i++;
30       }
31     } catch (InterruptedException exception) {}
32     Benchmark.N_PUTS.addAndGet(i);
33     Benchmark.SEMAPHORE.release();
34 } }
```

Figure 4.24: Java code for the performance experiments in this thesis (II)

subprograms (`BenchmarkProducer` and `BenchmarkConsumer`), where, in contrast to the previous examples, I use the interrupt mechanism of `put`/`get` to temporarily break, and later `resume`, I/O operations. (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, a `BenchmarkProducer` (`BenchmarkConsumer`) first performs a number of `put`s (`get`s) to warm up the JVM, until it gets `interrupted` by the main thread (i.e., after the main thread decides that the JVM had enough time to warm up, based on user input, not shown). Subsequently, again until it gets `interrupted`, the `BenchmarkProducer` (`BenchmarkConsumer`) performs a number of `put`s (`get`s), while it keeps count of its completed `put`s (`get`s). After the interrupt, the `BenchmarkProducer` (`BenchmarkConsumer`) adds its local count to a global count of all completed `put`s (`get`s). Note that a `BenchmarkProducer` not really produces actual data but always exchanges `0` through its `Port`. Similarly, a `BenchmarkConsumer` simply ignores all data it exchanges through its `Port`.

```
1  public class BenchmarkConsumer extends Thread {
2    private Datum datum;
3    private InputPort port;
4
5    public BenchmarkConsumer(InputPort port) {
6      this.port = port;
7    }
8
9    @Override
10   public void run() {
11     warmUp();
12     try {Benchmark.BARRIER.await();} catch (Exception exc) {System.exit(1);}
13     measure();
14   }
15
16   private void warmUp() {
17     Benchmark.SEMAPHORE.release();
18     try {
19       while (!Thread.interrupted()) port.get();
20     } catch (InterruptedException exception) {}
21   }
22
23   private void measure() {
24     int i = 0;
25     try {
26       port.resume();
27       while (!Thread.interrupted()) {
28         port.get();
29         i++;
30       }
31     } catch (InterruptedException exception) {}
32     Benchmark.N_GETS.addAndGet(i);
33     Benchmark.SEMAPHORE.release();
34 } }
```

Figure 4.25: Java code for the performance experiments in this thesis (III)

I used the classes in Figures 4.23–4.25 in the experiments in the next subsection, where I measure the performance of FOCAML-to-Java-compiled protocol subprograms through the number of completed puts/gets by `BenchmarkProducers`/`BenchmarkConsumers`; note that `BenchmarkProducers` and `BenchmarkConsumers` remain oblivious to the actual protocol among them—they merely see their own `Port`—so I can conveniently (re)use them, without modifications, with any protocol instantiated in the main thread.

Experiments I: Protocols

As a first performance evaluation, and primarily to find weak spots of Lykos that require improvement—so, definitely *not* to show what perfect code Lykos generates—I performed a number of experiments. For these experiments, I selected the families of constraint automata defined in Figure 3.12 (namely SyncK, FifoK, Merger, Router, LateAsyncMerger, and EarlyAsyncMerger), OddFibonacci,

and Chess, all introduced and described in Chapter 3. The natural number parameters of these families enable me to study the *scalability* of code generated for their members as the value of their parameter increases. Henceforth, I denote this parameter by k, for all families. For SyncK and FifoK, parameter k controls the number of SyncKs/FifoKs. For Merger, LateAsyncMerger and EarlyAsyncMerger, parameter k controls the number of producers. For Router and OddFibonacci, parameter k controls the number of consumers. For Chess, parameter k controls the number of chess engines.

For every selected family, I ran Lykos under a five-minute timeout from an Eclipse instance with 2048 MB of memory to generate code for the following twelve values of k:

$$1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64$$

In total, thus, I tried to generate 96 protocol implementations. Subsequently, I ran every piece of generated code five times on a machine with 24 cores (two Intel E5-2690V3 processors in two sockets), without Hyper-Threading and without Turbo Boost (i.e., with a static clock frequency). To measure the performance of only the compiler-generated code, I used computationally empty producers and consumers (very similar to those in Figure 1.10). In each run, then, I measured the number of *rounds* that every `Protocol` could complete in four minutes of execution time after warming up the Java virtual machine for thirty seconds.

For members of the SyncK family, every round consists of a synchronous `put`/`get` by the producer/consumer. This requires firing one transition. For members of the FifoK family, every round consists of a `put` by the producer followed by an asynchronous `get` by the consumer. This requires firing two transitions. For members of the Merger family, every round consists of a `put`/`get` by one of the producers/the consumer. This requires firing one transition. For members of the Router family, every round consists of a `put`/`get` by the producer/one of the consumers. This requires firing one transition. For members of both the LateAsyncMerger family and the EarlyAsyncMerger family, every round consists of a `put` by one of the producers followed by an asynchronous `get` by the consumer. This requires firing two transitions. For members of the OddFibonacci family, every round consists of a `put` by the producer and, in case of an odd Fibonacci number, an additional synchronous `get` by each of the consumers. Either case requires firing one transition. For members of the Chess family, every round consists of a full cycle through the constraint automaton in Figure 3.28. This requires firing four transitions.

Figure 4.26 shows the per-family performance charts, averaged over five runs. The solid lines represent the actual measurements; the dotted lines represent *inverse-proportional growth* with respect to $k = 1$. I adopt inverse-proportionality as an elementary point of reference, because it constitutes a critical threshold for scalability: if performance drops below inverse-proportional growth, performance deteriorates faster in k than k itself (e.g., doubling k more than halves the number of completed rounds). Importantly, the inverse-proportional curves in Figure 4.26 merely indicate a *lower bound* to good scalability:

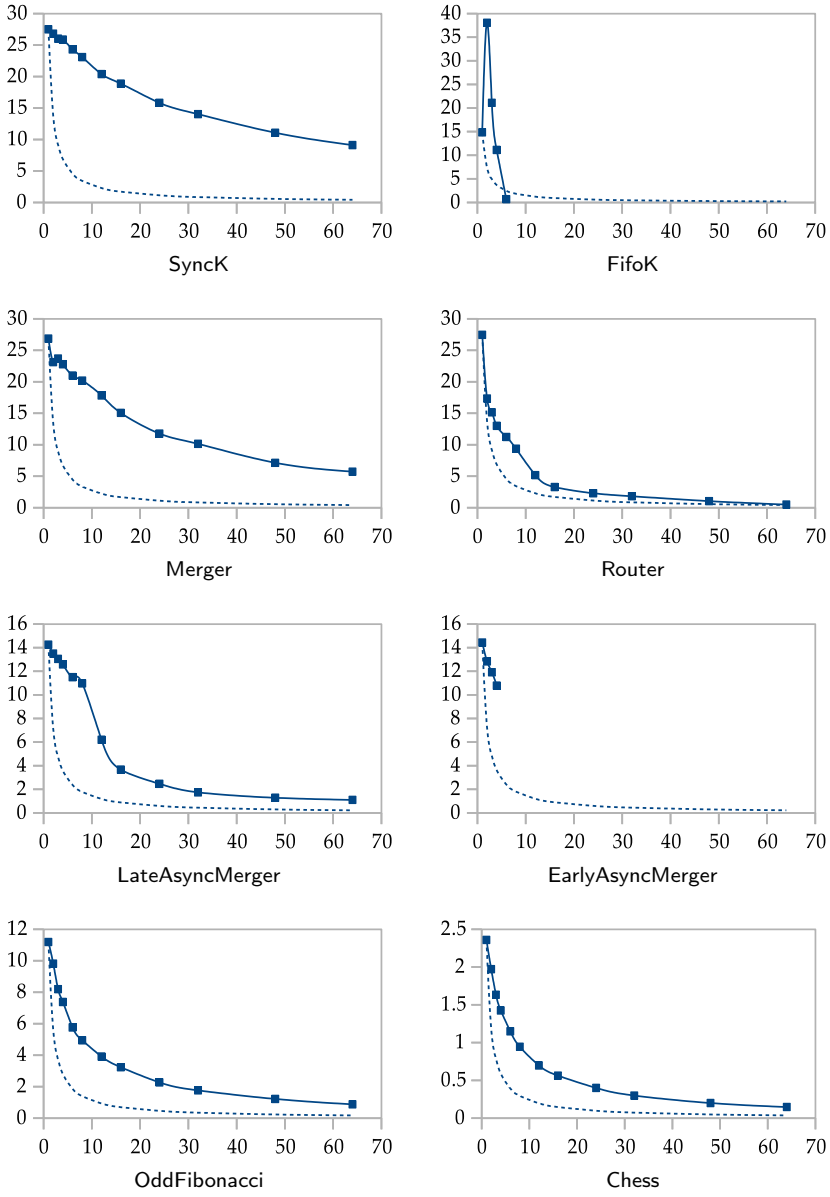


Figure 4.26: Performance (in number of completed rounds per four minutes) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See also the legend in Figure 9.1.

although growth below inverse-proportionality definitely indicates poor scalability, growth above inverse-proportionality does not necessarily imply good scalability—depending on the protocol, good scalability has stronger requirements. Thus, inverse-proportionality forms a necessary condition for good scalability but not necessarily a sufficient one. I come back to this point extensively in Chapter 6, where I also explain in more detail the relation between inverse-proportionality and good/poor scalability for the families with which I experimented in this chapter.

Figure 4.26 shows no measurements for members of $\mathsf{FifoK}_{>6}$ and $\mathsf{EarlyAsyncMerger}_{>4}$: Lykos exceeded its available resources trying to compile these members and thus failed to generate code. Moreover, after peeking at $k = 2$, the performance of the code generated for members of FifoK degrades rapidly as k increases. In fact, the performance measured for the FifoK_6 member lies below the critical threshold of inverse-proportionality, which indicates a serious scalability problem. In Chapter 5, I study both these compile-time and run-time problems.

Experiments II: Programs

With the FOCAML implementation of NPB presented in Chapter 3, I performed a second series of experiments to also evaluate the performance of code generated by Lykos in full programs and gain more insight. More precisely, I experimented with two FOCAML versions of every benchmark in the Java implementation of NPB: one that imposes an order (i.e., a rather literal translation of the Java implementation of NPB) and one that does not (i.e., a less literal but still intention-preserving version), as explained in Chapter 3. In experiments of the former kind, I evaluate members of MasterWorkersInteractionPatternA (all benchmarks except NPB-LU) and RelayRaceInteractionPatternA (NPB-LU); in experiments of the latter kind, I evaluate members of MasterWorkersInteractionPatternB (all benchmarks except NPB-LU) and RelayRaceInteractionPatternB (NPB-LU). As in the previous subsection, the natural number parameters of these families enable me to study the scalability of compiler generated-code, this time in the number of slaves.

For every benchmark and every version, I ran Lykos under a five-minute timeout from an Eclipse instance with 4096 MB of memory to generate code for the following six values of k:

$$2, 4, 8, 16, 32, 64$$

In total, thus, I *tried* to generate 84 full programs. Lykos, however, failed for all k, for all benchmarks, for all versions. Essentially, the constraint automata in these benchmarks suffer from the same compile-time problem as members of FifoK and EarlyAsyncMerger in the previous subsection. Here, however, this problem manifests already with the smallest value of k under consideration. I explain this problem in more detail in Chapter 5 and, fortunately, provide a solution as well.

Chapter 5

Improved Compilation I: Local Multiplication

The experimental results in Chapter 4 show that the Centralized Approach has two serious scalability problems. One of these problems manifests at compile-time (i.e., the inability to generate code for $\mathsf{Fifo}_{>6}$, $\mathsf{EarlyAsyncMerger}_{>4}$, and all FOCAML versions of the NPB benchmarks); the other manifests at run-time (i.e., the rapid performance degradation of Fifo_k as k increases, below inverse-proportionality).

In this chapter, to solve these problems, I develop a new compilation approach, which strikes a middle ground between the Distributed Approach and the Centralized Approach. In Section 5.1, I first provide a general description of this new compilation approach. Subsequently, I present the technical details of its most important new element: the computation of a partition of a set of constraint automata. These technicalities involve the introduction of a new multiplication and require a thorough study of the circumstances in which the old multiplication in Definition 29 coincides with this new multiplication. In Section 5.2, I present an improved version of Lykos using this new compilation approach, including new experimental results on performance.

Although the compilation approach presented in this chapter eventually results in improved compiler-generated code, I define this improvement at the higher level of constraint automata instead of at the lower level of GPL code. Not only does this facilitate more elegant formal reasoning about correctness (compared to reasoning directly about GPL code), but it also eases the automatic application of this improvement by a FOCAML compiler. Moreover, it makes this improvement independent of GPLs—Java in this thesis—so that the same optimization automatically applies to, for instance, generated C code.

5.1 Theory

*(With Arbab and Santini, I previously published fragments of the material in this section in conference/workshop papers [**J**A13a, **J**A14, **J**SA14] and journal papers [**J**A16, **J**SA15].)*

Hybrid Approach

I start by explaining the two scalability problems with the Centralized Approach in more detail. First, to explain its compile-time problem, I repeat the following observation from Chapter 4:

> "[...] in the Centralized Approach, compilation requires many resources [...] while execution requires few."

As the experimental results in Chapter 4 show, the Centralized Approach requires not *just* many resources but often *too* many resources. For instance, members of the $\mathsf{EarlyAsyncMerger}_k$ subfamily, defined in Figure 3.12, have as many as 2^k states. Each of those states models a permutation of the emptiness/fullness of k memory cells. Similarly, members of the FifoK_k subfamily have k memory cells, each of which can—at any instant—have content or not, yielding an exponentially-sized state space. Members of the $\mathsf{LateAsyncMerger}_k$ subfamily, in contrast, do not have exponentially-sized state spaces, because they have only one memory cell shared by all k producers (cf. one memory cell for every producer as members of EarlyAsyncMerger have). In any case, members of $\mathsf{EarlyAsyncMerger}_{256}$ have more states (roughly 10^{77}) than the observable universe has hydrogen atoms (overestimate: 10^{80}). Although anecdotal, this comparison "proves" not only the practical intractability of compiling instantiated family signatures such as `EarlyAsyncMerger(A[1..256],B)` under the Centralized Approach today but also its theoretical impossibility forever. Because 256 producers seems not unreasonably many, solving this *state space explosion problem* seems imperative.

To explain the run-time problem with the Centralized Approach, I repeat the following observation from Chapter 4:

> "a FOCAML compiler that generates code under the Centralized Approach yields protocol subprograms with low latency and low throughput."

Indeed, the compiler-generated protocol subprogram for a "large" automaton in the Centralized Approach defines exactly one protocol unit. Consequently, when generating code under the Centralized Approach, a FOCAML compiler effectively serializes all potential parallelism among the "small" automata that multiply into such a large automaton. My comparison between the Distributed Approach and the Centralized Approach in Chapter 4, involving `AliceBobCarolDave` in Figure 4.7, already showed such *oversequentialization*. Oversequentialization reduces throughput: in the worst case, the firing of one transition inhibits the parallel firing of other, completely independent transitions.

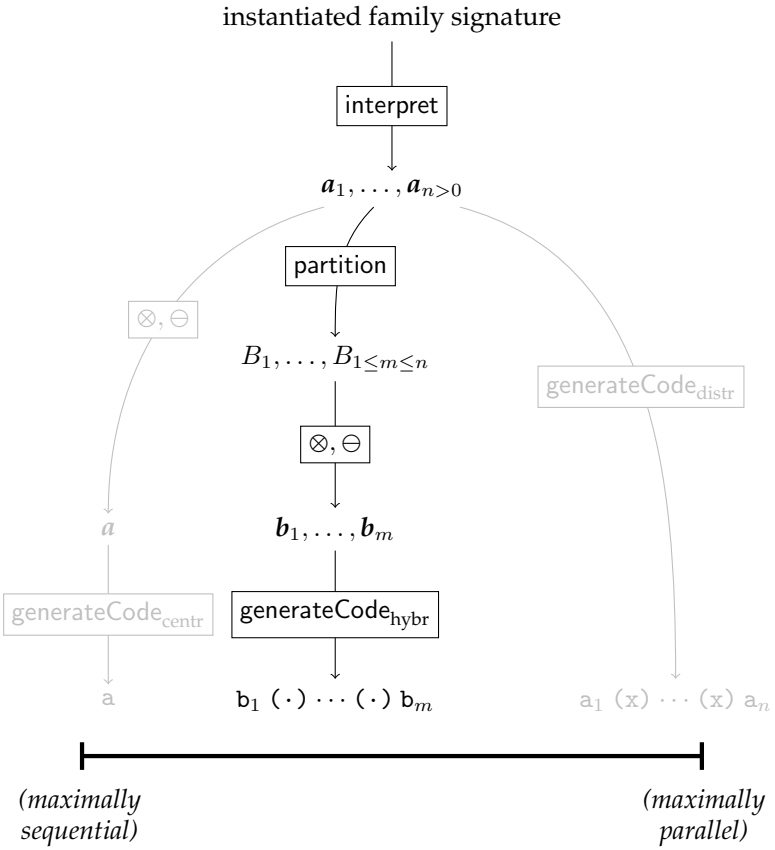


Figure 5.1: Hybrid compilation approach (cf. Figure 4.1)

As the number of independent transitions increases, this sequential bottleneck becomes an increasingly pressing problem.

To solve these problems of state space explosion and oversequentialization, I propose to adopt a new approach for FOCAML compilation: the *Hybrid Approach*, shown in Figure 5.1. On the sequentiality/parallelism-spectrum, the Hybrid Approach sits somewhere between the Centralized Approach and the Distributed Approach. The Hybrid Approach strikes a middle ground between those two ends: it sequentializes all *useless* parallelism to preserve only *useful* parallelism. I define useless/useful parallelism as follows.

- Protocol units exhibit useless parallelism whenever these protocol units must reach consensus about their global behavior before any of them can fire a local transition. In those cases, parallelism *does not* improve throughput, while the communication overhead of reaching consensus *does* reduce latency. Protocol units Alice, Bob, and Carol in the `AliceBobCarolDave` example in Chapter 4 illustrate useless parallelism: neither

Alice, nor Bob, nor Carol can fire a local transition without first having to communicate with the others.

Generally, protocol units that exhibit useless parallelism require an *expensive consensus algorithm*, with global communication among everyone. Previously, in Chapter 4, I denoted this algorithm by `(x)` and observed that this algorithm effectively multiplies constraint automata at run-time, in the sense of Definition 29.

- Protocol units exhibit useful parallelism whenever protocol units in one subset can fire local transitions independently of protocol units in another subset. In those cases, parallelism truly improves throughput. The subset of protocol units consisting of Alice, Bob, and Carol and the subset consisting of only Dave in the `AliceBobCarolDave` example illustrate useful parallelism: Dave can fire a local transition without first having to communicate with Alice, Bob, or Carol.

  Generally, protocol units that exhibit useful parallelism require a *cheap consensus algorithm*, with only local communication between neighbors. Henceforth, in this chapter, I denote this algorithm by `(·)` and argue that also this algorithm effectively multiplies constraint automata at run-time, albeit under a different—yet equivalent—multiplication.

Seen from the perspective of the Centralized Approach, the Hybrid Approach yields protocol subprograms that define $m \geq 1$ protocol units instead of just one (to preserve only useful parallelism); seen from the perspective of the Distributed Approach, the Hybrid Approach yields protocol subprograms that define $m \leq n$ protocol units instead of n (to sequentialize all useless parallelism). In cases of only useless parallelism, the Hybrid Approach reduces to the Centralized Approach (i.e., $m = 1$); in cases of only useful parallelism, the Hybrid Approach reduces to the Distributed Approach (i.e., $m = n$).

A FOCAML compiler that generates code under the Hybrid Approach takes four steps to generate GPL code on input of an instantiated family signature. In the first step, the compiler obtains a list of n small primitive constraint automata $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ in the same way as in the Distributed Approach and the Centralized Approach, by calling a FOCAML interpreter. In the second step, the compiler *partitions* the set of those small automata into m disjoint subsets $B_1, \ldots, B_m$. In particular, it computes a *reasonable partition*, where:

- protocol units for constraint automata in the same subset exhibit useless parallelism, while
- protocol units for constraint automata in different subset exhibit useful parallelism.

In the third step, the compiler multiplies the small automata in every part B_i and subtracts all internal ports, effectively serializing the useless parallelism among those automata. This step yields m "medium" composite constraint automata $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_m$. In the fourth step, the compiler translates these medium

automata to a protocol subprogram $\mathtt{b}_1\ \mathtt{(\cdot)} \cdots \mathtt{(\cdot)}\ \mathtt{b}_m$ in the GPL. This protocol subprogram defines m protocol units. Individually, every one of these protocol units locally simulates a medium automaton $\boldsymbol{b}_i$; collectively, these protocol units globally simulate the product of $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_m$. As in the Distributed Approach, to achieve the latter, the protocol units need to synchronize their local behavior with each other. Contrasting the Distributed Approach, however, by the construction of the reasonable partition in the second step, this synchronization needs to happen only locally between neighbors instead of globally among everyone. In Figure 5.1, I denote the code of the corresponding cheap consensus algorithm by `(·)`, placed between $\mathtt{b}_1, \ldots, \mathtt{b}_n$ to emphasize that this algorithm effectively multiplies constraint automata, just as the expensive consensus algorithm denoted by `(x)` does.

I discuss the event-handlers for protocol units that simulate medium automata later in this section. For now, suffice it to say that such event-handlers work roughly the same as the event-handler in Figure 4.5.

L-Multiplication

The main challenge with the Hybrid Approach lies in its second step: computing a reasonable partition in a potentially huge search space. After all, the number of unique partitions of a k-cardinality set grows superexponentially in k [Kla10]. In one corner of this search space, by putting every $\boldsymbol{a}_i$ in its own subset to get a partition $\{\{\boldsymbol{a}_1\}, \ldots, \{\boldsymbol{a}_n\}\}$, the Hybrid Approach reduces to the Distributed Approach; in its opposite corner, by putting every $\boldsymbol{a}_i$ in the same subset to get a partition $\{\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}\}$, the Hybrid Approach reduces to the Centralized Approach. Typically, however, neither of these two corners yields a reasonable partition. For now, I temporarily park the issue of computing reasonable partitions. Instead, at this point, just out of scientific curiosity, I study under which circumstances substituting `(x)` with `(·)` "preserves the original behavior" (i.e., under which circumstances synchronizing the behavior of protocol units with the cheap consensus algorithm instead of the expensive one preserves their original behavior, i.e., under which circumstances only local communication between neighbors can safely replace global communication among everyone). Incidentally, the insight resulting from this investigation yields an algorithm for computing reasonable partitions as well.

The previous observation that `(x)` applies $\otimes$ at run-time implies that—from the opposite perspective—$\otimes$ models `(x)`. Interestingly, I can similarly define *another* multiplication that models `(·)`. After doing so, instead of studying the interchangeability of `(x)` and `(·)` at the practical level of protocol units and their consensus algorithms, I can more conveniently study it at the theoretical level of constraint automata and their multiplications.

The "new" multiplication differs from the "old" multiplication only in how new transition relations come about. With the old multiplication in Definition 29, $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ synchronously fire transitions that agree on the involvement of shared ports under a rather weak notion of agreement: in addition to their shared ports, $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ allow each other to involve also any number of un-

shared ports. This weak agreement, formalized in Definition 27 of $\Diamond$, gives rise to powerful multiparty and indirect synchronization as explained in Chapter 2. Exactly those properties, however, make `(x)` expensive: whenever a transition in constraint automaton $\boldsymbol{b}_1$ and a transition in constraint automaton $\boldsymbol{b}_k$ synchronously fire via transitions in constraint automata $\boldsymbol{b}_2, \ldots, \boldsymbol{b}_{k-1}$, at run-time, their protocol units must globally communicate to ensure that the protocol unit for $\boldsymbol{b}_1$ actually reaches consensus with the protocol unit for $\boldsymbol{b}_k$. With only local communication between neighbors, as stipulated for `(·)`, such multiparty and indirect synchronization cannot happen. Therefore, to model `(·)`, the new multiplication must restrict these forms of synchronization by strengthening the previous weak notion of agreement. Essentially, under the resulting *strong agreement*, constraint automata forbid each other to involve unshared ports in their synchronously firing transitions, whereas under the previous *weak agreement*, constraint automata allow each other to do so (which ultimately gives rise to multiparty and indirect synchronization). More precisely, under strong agreement, transitions in $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ can synchronously fire only if one of those transitions involves only shared ports, or if both transitions involve only unshared ports (i.e., at run-time, their protocol units communicate either only locally or not whatsoever).

Definition 31 (strong agreement)**.** $\blacklozenge \subseteq (2^{\mathbb{P}} \times 2^{\mathbb{P}}) \times (2^{\mathbb{P}} \times 2^{\mathbb{P}})$ *denotes the smallest relation induced by the following rule:*

$$\frac{P_1 \subseteq P_1^{\text{all}} \ \textbf{and} \ P_2 \subseteq P_2^{\text{all}} \ \textbf{and} \begin{bmatrix} P_1^{\text{all}} \cap P_2 = P_2^{\text{all}} \cap P_1 = \emptyset \\ \textbf{or} \ \ P_1 = P_1^{\text{all}} \cap P_2 \\ \textbf{or} \ \ P_2 = P_2^{\text{all}} \cap P_1 \end{bmatrix}}{(P_1^{\text{all}}, P_1) \blacklozenge (P_2^{\text{all}}, P_2)} \tag{5.1}$$

The following lemma states that strong agreement implies weak agreement. This, combined with Lemma 1, also means that $\blacklozenge$ actually constitutes an agreement relation in the sense of Definition 26.

Lemma 2. $(P_1^{\text{all}}, P_1) \blacklozenge (P_2^{\text{all}}, P_2)$ **implies** $(P_1^{\text{all}}, P_1) \Diamond (P_2^{\text{all}}, P_2)$

Lemma 3. $\blacklozenge \in \mathbb{A}\text{GREEM}$

The new multiplication, henceforth called *l(ocal)-multiplication*, consumes two constraint automata $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ as input and produces a new constraint automaton as output in nearly the same way as the old multiplication, henceforth called *g(lobal)-multiplication*: as g-multiplication, I define l-multiplication by instantiating the generalized multiplication in Definition 28 with strong agreement (instead of with weak agreement as in Definition 29 of $\otimes$).

Definition 32 (l-multiplication)**.** $\odot : \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM} \rightharpoonup \mathbb{A}\text{UTOM}$ *denotes the partial function defined by the following equation:*

$$\boldsymbol{a}_1 \odot \boldsymbol{a}_2 = \boldsymbol{a}_1 \circledast_{\blacklozenge} \boldsymbol{a}_2$$

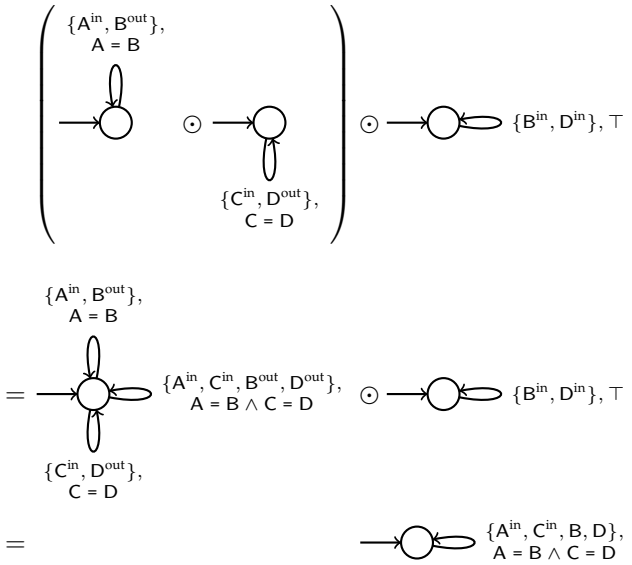


Figure 5.2: Left-associative l-multiplication of $\mathsf{Sync}(\mathsf{A};\mathsf{B})$, $\mathsf{Sync}(\mathsf{C};\mathsf{D})$, and $\mathsf{SyncDrain}(\mathsf{B},\mathsf{D};)$

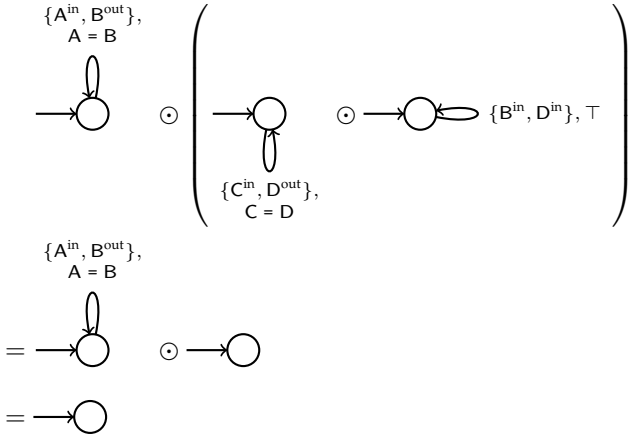


Figure 5.3: Right-associative l-multiplication of $\mathsf{Sync}(\mathsf{A};\mathsf{B})$, $\mathsf{Sync}(\mathsf{C};\mathsf{D})$, and $\mathsf{SyncDrain}(\mathsf{B},\mathsf{D};)$

L-multiplication satisfies commutativity and idempotence (up-to behavioral congruence), but in contrast to g-multiplication, it does not satisfy associativity: generally, $(\boldsymbol{a}_1 \odot \boldsymbol{a}_2) \odot \boldsymbol{a}_3 \not\approx \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3)$. Figures 5.2 and 5.3 exemplify this phenomenon: both the l-product in Figure 5.2 and the l-product in Figure 5.3 have one state, but the former l-product has one transition, whereas the

latter l-product has no transitions. I postpone a more detailed discussion of l-multiplication's nonassociativity until later in this section. To minimize numbers of parentheses, I assume right-associative notation for $\odot$. For instance, I write $\boldsymbol{a}_1 \odot \boldsymbol{a}_2 \odot \boldsymbol{a}_3 \odot \boldsymbol{a}_4$ for $\boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_3 \odot \boldsymbol{a}_4))$. The following theorem states that $\simeq$ denotes a congruence under $\odot$.

Theorem 6. $\begin{bmatrix} \boldsymbol{a}_1 \odot \boldsymbol{a}_3, \boldsymbol{a}_2 \odot \boldsymbol{a}_4 \in \mathbb{A}\text{UTOM} \\ \textbf{and } \boldsymbol{a}_1 \simeq \boldsymbol{a}_2 \textbf{ and } \boldsymbol{a}_3 \simeq \boldsymbol{a}_4 \end{bmatrix}$ **implies** $\boldsymbol{a}_1 \odot \boldsymbol{a}_3 \simeq \boldsymbol{a}_2 \odot \boldsymbol{a}_4$

First Characterization

The fact that g-multiplication satisfies associativity, whereas l-multiplication does not, already implies that substituting $\otimes$ with $\odot$ not always "preserves the original behavior" (in the sense previously explained). For instance, the g-product of the constraint automata in Figure 5.3 (under the same placement of parentheses) equals the l-product in Figure 5.2 but not the l-product in Figure 5.3. To determine when substituting $\otimes$ with $\odot$ preserves the original behavior, I first study under which conditions a g-product of two constraint automata simulates their l-product and vice versa.

Lemma 2 immediately implies that $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2$ simulates $\boldsymbol{a}_1 \odot \boldsymbol{a}_2$. In other words, a g-product of two constraint automata has at least the same transitions as their l-product. At run-time, this means that protocol units that use a consensus algorithm with global communication can effectuate at least the same instances of interaction as those that use a consensus algorithm with only local communication. This makes perfect sense. The inverse statement, in contrast, does not: protocol units that use a consensus algorithm with only local communication may not effectuate the same instances of interaction as those that use a consensus algorithm with global communication. In terms of constraint automata, this corresponds to the fact that $\boldsymbol{a}_1 \odot \boldsymbol{a}_2$ not necessarily simulates $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2$. Indeed, if transitions of $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ agree on the involvement of their shared ports (which $\otimes$ requires), this does not necessarily mean that they involve no other ports (which $\odot$ additionally requires). To characterize the cases in which it does, I define *conditional strong agreement* as a relation "between" $\blacklozenge$ and $\lozenge$ (and lifted from transitions to constraint automata): $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ conditionally strongly agree iff, for each of their transitions, their weak agreement on the involvement of their shared ports implies their strong agreement.

Definition 33 (Conditional strong agreement)**.** $\langle\!\blacklozenge \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\left[\begin{bmatrix} q_1 \xrightarrow{P_1,\phi_1}_1 q_1' \textbf{ and } q_2 \xrightarrow{P_2,\phi_2}_2 q_2' \\ \textbf{and } (P_1^{\text{all}}, P_1) \lozenge (P_2^{\text{all}}, P_2) \end{bmatrix} \textbf{ implies } (P_1^{\text{all}}, P_1) \blacklozenge (P_2^{\text{all}}, P_2)\right] \quad \textbf{for all } q_1, q_2, q_1', q_2', P_1, P_2, \phi_1, \phi_2}{(\cdot, (P_1^{\text{all}}, P_1^{\text{in}}, P_1^{\text{out}}), \cdot, \longrightarrow_1, \cdot) \langle\!\blacklozenge (\cdot, (P_2^{\text{all}}, P_2^{\text{in}}, P_2^{\text{out}}), \cdot, \longrightarrow_2, \cdot)} \tag{5.2}$$

The following lemma states that conditional strong agreement between two constraint automata implies the behavioral congruence of their l-product and their g-product.

Lemma 4. $\left[\boldsymbol{a}_1 \mathbin{\langle\!\blacklozenge\!\rangle} \boldsymbol{a}_2 \text{ and } \boldsymbol{a}_1 \odot \boldsymbol{a}_2 \in \mathbb{A}\text{UTOM}\right]$ **implies** $\boldsymbol{a}_1 \odot \boldsymbol{a}_2 \simeq \boldsymbol{a}_1 \otimes \boldsymbol{a}_2$

As a generalization of Lemma 4, suppose that I have a list of k constraint automata such that every i-th constraint automaton in this list conditionally strongly agrees with the l-product of all higher positioned ones. The order of the constraint automata matters, because $\odot$ does not exhibit associativity. The following theorem states that multiplying all constraint automata in the previous list with $\odot$ or $\otimes$, starting from the ones in the highest positions (i.e., the most deeply nested ones under right-associative notation), yields behaviorally congruent products.

Theorem 7.

$$\left[\begin{array}{l}\left[\begin{array}{l}1 \leq i < k \text{ implies} \\ \boldsymbol{a}_i \mathbin{\langle\!\blacklozenge\!\rangle} \boldsymbol{a}_{i+1} \odot \cdots \odot \boldsymbol{a}_k\end{array}\right] \text{ for all } i \\ \text{and } \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_k \in \mathbb{A}\text{UTOM}\end{array}\right] \text{ implies } \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_k \simeq \boldsymbol{a}_1 \otimes \cdots \otimes \boldsymbol{a}_k$$

I call the premise in the previous theorem the $\langle\!\blacklozenge\!\rangle$*-based characterization* of when substituting $\otimes$ with $\odot$ preserves the original behavior.

The $\langle\!\blacklozenge\!\rangle$-based characterization has two disadvantages. First, to test if two constraint automata $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ conditionally strongly agree, one must pairwise compare their transitions. By itself, this may already require a significant amount of computation (i.e., $\mathcal{O}(k_1 k_2)$ for $k_1 = |\mathsf{Trans}(\boldsymbol{a}_1)|$ and $k_2 = |\mathsf{Trans}(\boldsymbol{a}_2)|$). Moreover, the $\langle\!\blacklozenge\!\rangle$-based characterization requires conditional strong agreement not between individual constraint automata in the list but between their l-products. Because l-multiplication generally does not preserve conditional strong agreement, checking a list of constraint automata for satisfaction of the $\langle\!\blacklozenge\!\rangle$-based characterization requires the potentially expensive computation of many l-products. These disadvantages make the $\langle\!\blacklozenge\!\rangle$-based characterization unattractive in practice. In the next subsection, I therefore study a cheaper characterization.

Cheaper Characterization

I develop a cheaper characterization through a number of observations. First, using only local communication between protocol units instead of global communication clearly preserves the original behavior of *independent* protocol units that do not need to communicate with each other whatsoever. Thus, substituting $\otimes$ with $\odot$ should preserve the original behavior at least when applied to constraint automata corresponding to such independent protocol units. I start by formally defining the notion of independence.

Definition 34 (independence). $\asymp \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{P_1^{\text{all}} \cap P_2^{\text{all}} = \emptyset}{(\cdot, (P_1^{\text{all}}, \cdot, \cdot), \cdot, \cdot, \cdot) \asymp (\cdot, (P_2^{\text{all}}, \cdot, \cdot), \cdot, \cdot, \cdot)} \tag{5.3}$$

The following lemmas state (i) that independent constraint automata conditionally strongly agree with each other and (ii) that l-multiplication preserves independence.

Lemma 5. $\boldsymbol{a_1 \asymp a_2}$ **implies** $\boldsymbol{a_1 \blacklozenge a_2}$

Lemma 6. $\left[\boldsymbol{a_1 \odot a_2 \in \mathbb{A}\text{UTOM}}\right.$ **and** $\left.\boldsymbol{a \asymp a_1, a_2}\right]$ **implies** $\boldsymbol{a \asymp a_1 \odot a_2}$

Lemmas 5 and 6 and Theorem 7 imply that substituting $\otimes$ with $\odot$ preserves the original behavior if their multiplicands satisfy independence. Moreover, checking for independence requires fewer resources than checking for conditional strong agreement, namely $\mathcal{O}(1)$ instead of $\mathcal{O}(k_1 k_2)$.

Although checking constraint automata for independence costs close to nothing, the result implied by Lemmas 5 and 6 and Theorem 7 in its present form has limited practical value: independent constraint automata only rarely occur outside artificial examples. To get a more useful result, I introduce the notion of *slavery* and afterward combine it with independence. I start by formally defining when a constraint automaton $\boldsymbol{a_2}$ has enslaved a constraint automaton $\boldsymbol{a_1}$. In that case, every transition in $\boldsymbol{a_1}$ that involves *some* port shared with $\boldsymbol{a_2}$, involves *only* ports shared with $\boldsymbol{a_2}$. In other words, $\boldsymbol{a_2}$ completely dictates what $\boldsymbol{a_1}$ does whenever a transition in $\boldsymbol{a_1}$ involves at least one of their shared ports. Importantly, this notion of slavery does not forbid $\boldsymbol{a_1}$ from firing transitions that involve only ports that $\boldsymbol{a_2}$ does not know about (i.e., slaves can secretly "rebel"). This enables other constraint automata to enslave $\boldsymbol{a_1}$ as well (albeit with respect to different ports).

Definition 35 (slavery). $\mapsto \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\left[\begin{bmatrix} q_1 \xrightarrow{P_1, \phi_1}_1 q_1' \\ \textbf{and } P_1 \cap P_2^{\text{all}} \neq \emptyset \end{bmatrix} \textbf{ implies } P_1 \subseteq P_2^{\text{all}}\right] \textbf{ for all } q_1, q_1', P_1, \phi_1}{(\cdot, \cdot, \cdot, \longrightarrow_1, \cdot) \mapsto (\cdot, (P_2^{\text{all}}, \cdot, \cdot), \cdot, \cdot, \cdot)} \tag{5.4}$$

The following lemmas state that if one constraint automaton has enslaved another constraint automaton, they conditionally strongly agree with each other and that l-multiplication preserves slavery.

Lemma 7. $\boldsymbol{a_1 \mapsto a_2}$ **implies** $\boldsymbol{a_1 \blacklozenge a_2}$

Lemma 8. $\begin{bmatrix} \boldsymbol{a}_1 \odot \boldsymbol{a}_2 \in \mathbb{A}\text{UTOM} \textbf{ and } \boldsymbol{a} \mapsto \boldsymbol{a}_1 \\ \textbf{and } \left[\boldsymbol{a} \asymp \boldsymbol{a}_2 \textbf{ or } \boldsymbol{a} \mapsto \boldsymbol{a}_2\right] \end{bmatrix}$ **implies** $\boldsymbol{a} \mapsto \boldsymbol{a}_1 \odot \boldsymbol{a}_2$

Lemma 9. $\begin{bmatrix} \boldsymbol{a}_1 \odot \boldsymbol{a}_2 \in \mathbb{A}\text{UTOM} \textbf{ and } \boldsymbol{a} \mapsto \boldsymbol{a}_2 \\ \textbf{and } \left[\boldsymbol{a} \asymp \boldsymbol{a}_1 \textbf{ or } \boldsymbol{a} \mapsto \boldsymbol{a}_1\right] \end{bmatrix}$ **implies** $\boldsymbol{a} \mapsto \boldsymbol{a}_1 \odot \boldsymbol{a}_2$

Lemmas 7, 8, and 9 and Theorem 7 imply that substituting $\otimes$ with $\odot$ preserves the original behavior if their multiplicands satisfy slavery. Moreover, checking for slavery requires fewer resources than checking for conditional strong agreement, namely $\mathcal{O}(k_1)$ instead of $\mathcal{O}(k_1 k_2)$.

By combining independence and slavery, I obtain *conditional slavery*.

Definition 36 (conditional slavery)**.** $\between \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\boldsymbol{a}_1 \not\asymp \boldsymbol{a}_2 \textbf{ implies } \boldsymbol{a}_1 \mapsto \boldsymbol{a}_2}{\boldsymbol{a}_1 \between \boldsymbol{a}_2} \tag{5.5}$$

The following lemmas state (i) that if one constraint automaton has conditionally enslaved another constraint automaton, they conditionally strongly agree and (ii) that l-multiplication preserves conditional slavery.

Lemma 10. $\boldsymbol{a}_1 \between \boldsymbol{a}_2$ **implies** $\boldsymbol{a}_1 \blacklozenge \boldsymbol{a}_2$

Lemma 11. $\left[\boldsymbol{a}_1 \odot \boldsymbol{a}_2 \in \mathbb{A}\text{UTOM} \textbf{ and } \boldsymbol{a} \between \boldsymbol{a}_1, \boldsymbol{a}_2\right]$ **implies** $\boldsymbol{a} \between \boldsymbol{a}_1 \odot \boldsymbol{a}_2$

Lemmas 10 and 11 and Theorem 7 imply that substituting $\otimes$ with $\odot$ preserves the original behavior if their multiplicands satisfy conditional slavery. Moreover, checking for conditional slavery costs the same as checking for slavery (i.e., less than checking for conditional strong agreement).

With conditional slavery, in contrast to independence alone, I can define a sufficiently powerful characterization of when substituting $\otimes$ with $\odot$ preserves the original behavior. Similar to Theorem 7, suppose that I have a list of k constraint automata such that every i-th constraint automaton in this list has conditionally enslaved all constraint automata in a lower position. The following theorem states that multiplying all constraint automata in the previous list with $\odot$ or $\otimes$, starting from the ones in the highest positions (i.e., the most deeply nested ones under right-associative notation), yields behaviorally congruent products.

Theorem 8.

$$\begin{bmatrix} \left[\begin{bmatrix} 1 \leq i < k \textbf{ implies} \\ \boldsymbol{a}_i \between \boldsymbol{a}_{i+1}, \ldots, \boldsymbol{a}_k \end{bmatrix} \textbf{ for all } i\right] \\ \textbf{and } \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_k \in \mathbb{A}\text{UTOM} \end{bmatrix} \textbf{ implies } \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_k \simeq \boldsymbol{a}_1 \otimes \cdots \otimes \boldsymbol{a}_k$$

I call the premise in the previous theorem the $\between$*-based characterization* of when substituting $\otimes$ with $\odot$ preserves the original behavior.

To give Theorem 8 a more natural interpretation, I strengthen its premise: the following theorem states that substituting $\otimes$ with $\odot$ preserves the original behavior whenever I have (i) k constraint automata conditionally enslaved by *all* other constraint automata and (ii) l pairwise independent "master" constraint automata.

Theorem 9.

$$\left[\left[\begin{bmatrix}1 \leq i \leq k \textbf{ implies}\\ \boldsymbol{a}_i \bowtie \boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i-1},\\ \qquad \boldsymbol{a}_{i+1}, \ldots, \boldsymbol{a}_{k+l}\end{bmatrix} \textbf{ and } \begin{bmatrix}k+1 \leq i \leq k+l \textbf{ implies}\\ \boldsymbol{a}_i \asymp \boldsymbol{a}_{k+1}, \ldots, \boldsymbol{a}_{i-1},\\ \qquad \boldsymbol{a}_{i+1}, \ldots, \boldsymbol{a}_{k+l}\end{bmatrix}\right] \textbf{ for all } i\right]$$
$$\textbf{and } \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_{k+l} \in \mathbb{A}\text{UTOM}$$
$$\textbf{implies } \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_{k+l} \simeq \boldsymbol{a}_1 \otimes \cdots \otimes \boldsymbol{a}_{k+l}$$

Because of their pairwise independence, protocol units for masters never *directly* communicate with each other. If multiple masters share the same slave, though, their protocol units may communicate *indirectly* with each other, via that slave. Such indirect communication occurs always asynchronously: otherwise, in case of synchronous communication, the slave would have a transition involving ports shared with multiple masters, which slavery forbids.

The previous interpretation of constraint automata as masters and slaves corresponds to the notion of synchronous and asynchronous *regions* in the Reo literature, perhaps first mentioned by Clarke et al. [CCA07]. Roughly, one can always split a Reo circuit into subcircuits—its regions—such that interaction on ports in such a subcircuit occurs always either asynchronously (i.e., every firing transition involves at most one port) or eventually synchronously (i.e., at least one firing transition involves more than one port). Circuits have *maximal* synchronous regions in the sense that no two synchronous regions have shared ports: every circuit has, by definition, only pairwise independent synchronous regions. Consequently, the constraint automata for the synchronous regions of a circuit can act as the l masters in Theorem 9. Dually, circuits have *minimal* asynchronous regions in the sense that no asynchronous region consists of more than one primitive. Asynchronous regions effectively constitute asynchronous communication mediums between synchronous regions. Consequently, the constraint automata for the asynchronous regions of a circuit can act as a the k conditional slaves in Theorem 9.

Synchronous and asynchronous regions play an important role in Reo compilers/interpreters that *split* circuits along the boundaries of their regions to decouple those regions' execution and improve performance. For his PhD thesis [Pro11], Proença developed the first implementation based on these ideas and invented a new automaton model to reason about split circuits [PCdVA11, PCdVA12]. Later, Clarke and Proença studied circuit splitting in the context of *coloring semantics* [CP12]. They discovered that the standard version of coloring semantics has undesirable properties in the context of circuit splitting: some split circuits that intuitively *should* behave as their originals nevertheless have inequivalent coloring semantics. To address this problem, Clarke

and Proença proposed a new variant of coloring semantics that better supports locality and independence. Before developing the Hybrid Approach as presented in this section, with Clarke and Proença, I worked on a formalization of circuit splitting in a process algebraic setting [**J**CP12, **J**CP16]. All this earlier work on circuit splitting strongly inspired and influenced me in developing the Hybrid Approach.

Practical Characterization

Although cheaper than the $\langle\!\blacklozenge$-based characterization, the $\bowtie$-based characterization still requires relatively many computational resources in practice. Therefore, I strengthen it once more by introducing another relation on constraint automata: *no-synchronization*. Informally, a constraint automaton exhibits no-synchronization if it never synchronizes any of its ports (i.e., each of its transitions has a singleton synchronization constraint).

Definition 37 (no-synchronization). $\xrightarrow{1} \subseteq \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\left[q \xrightarrow{P,\phi} q' \ \textbf{implies} \ |P| = 1\right] \ \textbf{for all} \ q, q', P, \phi}{\xrightarrow{1} (\cdot, \cdot, \cdot, \longrightarrow, \cdot)} \tag{5.6}$$

The following lemma states that no-synchronization implies conditional slavery.

Lemma 12. $\xrightarrow{1} \boldsymbol{a}_1 \ \textbf{implies} \ \boldsymbol{a}_1 \bowtie \boldsymbol{a}_2$

The following theorem follows from Lemma 12 and Theorem 9.

Theorem 10.

$$\left[\left[\begin{bmatrix} 1 \leq i \leq k \\ \textbf{implies} \ \xrightarrow{1} \boldsymbol{a}_i \end{bmatrix} \textbf{and} \begin{bmatrix} k+1 \leq i \leq k+l \ \textbf{implies} \\ \boldsymbol{a}_i \asymp \boldsymbol{a}_{k+1}, \ldots, \boldsymbol{a}_{i-1}, \\ \boldsymbol{a}_{i+1}, \ldots, \boldsymbol{a}_{k+l} \end{bmatrix}\right] \textbf{for all} \ i\right] \textbf{and} \ \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_{k+l} \in \mathbb{A}\text{UTOM}$$
$$\textbf{implies} \ \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_{k+l} \simeq \boldsymbol{a}_1 \otimes \cdots \otimes \boldsymbol{a}_{k+l}$$

I call the premise in the previous theorem the $\xrightarrow{1}$*-based characterization* of when substituting $\otimes$ with $\odot$ preserves the original behavior.

Now, recall from earlier in this section that I actually wanted to find an algorithm for computing reasonable partitions. Incidentally, the $\xrightarrow{1}$-based characterization yields such an algorithm, namely Algorithm 1. Algorithm 1 iterates over an input set of n constraint automata and terminates in $\mathcal{O}(n^2)$. In each iteration, either it puts the current constraint automaton $\boldsymbol{a}_i$ in a new subset (if $\boldsymbol{a}_i$ satisfies no-synchronization), or it computes a new subset for $\boldsymbol{a}_i$, possibly including existing parts, such that the new subset contains all constraint

Algorithm 1 Algorithm for partitioning a set of constraint automata A

Require: $A^{\text{small}} = \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}$

function ALGORITHM1(A^{small})
  $\mathcal{A}^{\text{sl}} := \emptyset$
  $\mathcal{A}^{\text{m}} := \emptyset$
  $i := 1$
  while $i \leq n$ **do**
    if $\xrightarrow{1} \boldsymbol{a}_i$ **then**
      $\mathcal{A}^{\text{sl}} := \mathcal{A}^{\text{sl}} \cup \{\{\boldsymbol{a}_i\}\}$
    else
      $\overline{\mathcal{A}} := \{\overline{A} \mid \overline{\boldsymbol{a}} \in \overline{A} \in \mathcal{A}^{\text{m}} \textbf{ and } \boldsymbol{a}_i \not\asymp \overline{\boldsymbol{a}}\}$
      $\mathcal{A}^{\text{m}} := (\mathcal{A}^{\text{m}} \setminus \overline{\mathcal{A}}) \cup \{\{\boldsymbol{a}_i\} \cup \bigcup \overline{\mathcal{A}}\}$
    $i := i + 1$
  return $\mathcal{A}^{\text{sl}} \cup \mathcal{A}^{\text{m}}$

Ensure:
$$\left[\begin{array}{l} \mathcal{A}^{\text{sl}} \cup \mathcal{A}^{\text{m}} \text{ denotes a partition of } A^{\text{small}} \\ \textbf{and } \mathcal{A}^{\text{sl}} = \{A_1, \ldots, A_k\} \textbf{ and } \mathcal{A}^{\text{m}} = \{A_{k+1}, \ldots, A_{k+l}\} \\ \textbf{and } \left[\left[1 \leq j \leq k \textbf{ implies } \xrightarrow{1} \bigotimes A_j\right] \textbf{ for all } j\right] \\ \textbf{and } \left[\left[\begin{array}{c} k+1 \leq j \leq k+l \textbf{ implies} \\ \bigotimes A_j \asymp \bigotimes A_{k+1}, \ldots, \bigotimes A_{j-1}, \\ \bigotimes A_{j+1}, \ldots, \bigotimes A_{k+l} \end{array}\right] \textbf{ for all } j\right] \\ \qquad \textbf{for some } A_1, \ldots, A_{k+l}, k, l \end{array}\right]$$

automata dependent—directly or indirectly—on $\boldsymbol{a}_i$. The following theorem states the algorithm's correctness: it yields a partition of the input set and this partition satisfies the premise in Theorem 10.

Theorem 11. *Algorithm 11 is correct.*

Recall that a partition qualifies as reasonable if protocol units for constraint automata in the same subset exhibit useless parallelism, while protocol units for constraint automata in different parts exhibit useful parallelism. With respect to the first requirement, because constraint automata in the same subset all depend—directly or indirectly—on each other, their protocol units exhibit useless parallelism. With respect to the second requirement, by ordering medium automata as in Theorem 10 (Theorem 11 guarantees the existence of a list so ordered), I can safely substitute all g-multiplications between those medium automata with l-multiplications. At run-time, because g- and l-multiplication model the expensive and the cheap consensus algorithm, the protocol units for those medium automata require only the cheap consensus algorithm instead of the expensive one and, as such, exhibit useful parallelism. Thus, the partition computed by Algorithm 1 indeed qualifies as reasonable.

The Hybrid Approach, and Algorithm 1 in particular, directly address the problem of oversequentialization as introduced in the beginning of this chapter. In contrast, the Hybrid Approach does not directly address the problem of state space explosion. Nevertheless, Algorithm 1 solves also this other problem *at least in this thesis*. To see this, recall from Figure 3.4 that this thesis' core set contains only one family of constraint automata with more than one state—Fifo—each of whose members satisfies no-synchronization. Consequently, every Fifo gets its own singleton part, while all nonsingleton parts contain only single-state primitives. Because the product of any number of single-state constraint automata has only one state as well, and because multiplication occurs only on a per-subset basis in the Hybrid Approach, no state space explosion can happen under the core set in Figure 3.4. Generally, if every family of constraint automata with more than one state in a core set consists only of members that satisfy no-synchronization, the Hybrid Approach has no state space explosion problem.

Suppose that I can construct every constraint automaton out of members of the families in Figure 3.4. Moreover, suppose that I have an algorithm for *decomposing* every constraint automaton into such instances. Then, regardless of the particular primitives in a core set, the Hybrid Approach *never* suffers from state space explosion. After all, in that case, I can decompose every constraint automaton with more than one state into a number of Fifos and a number of single-state primitives. For arbitrary constraint automata, at this point, I have evidence neither for nor against the feasibility of this approach. Other people have worked on this topic for particular classes of constraint automata, though: Arbab et al. and Baier et al. devised multiple algorithms for decomposing "original constraint automata" [ABdB+05, BKK14], Koehler and Clarke proved a decomposition theorem for port automata [KC09], while Pourvatan et al. developed a *division* for constraint automata with state memory [PSAB12].

Related Work on Distributed Coordination

Three decades ago, in the mid 1980s, Gelernter introduced the coordination language Linda [Gel85]. At the heart of Linda lies the concept of a *tuple space*, a structure in which both workers and tuples of data, originating from and accessible to those workers, "float". Although a tuple space gives the programmer the illusion of shared memory, at the hardware level, this memory may actually reside at n different locations. Several approaches to implementing physically distributed tuple spaces exist. For instance, one can maintain the entire tuple space at one of the n locations (e.g., Feng et al. [FWY96], Wyckoff et al. [WMLF98]), but although simple to implement, this does not scale well in the number of computation processes [FGY94]. The Centralized Approach presented in Chapter 4 actually has a similar scalability problem (due to oversequentialization). Alternatively, one can scatter (with or without replication) the tuples in the tuple space over all n locations. Although such an approach has better scalability, one must resolve several issues to obtain a workable implementation, such as deciding where to store which tuple, efficiently

retrieving tuples, and load balancing [PA98]. Examples include the work by Bjornson [Bjo93], Feng et al. [FGY94], Rowstron and Wood [RW96], Menezes and Tolksdorf [MT03], and Atkinson [Atk10]. Although both distributed tuple spaces and the Hybrid Approach facilitate a form of distributed coordination, they differ in one fundamental aspect: whereas distributed tuple spaces distribute data (i.e., tuples), the Hybrid Approach distributes control (i.e., medium automata).

Bonakdarpour et al. worked on an approach for automatically generating distributed implementations for specifications in BIP [BBJ+12], a framework for specifying component-based systems at three specification levels [BBS06]: behavior of components, interaction between components, and priorities on interaction. BIP forbids simultaneous execution of conflicting instances of interaction (i.e., instances that involve overlapping sets of ports). In automatically generated distributed implementations of BIP specifications, therefore, Bonakdarpour et al. have to ensure that such conflicting interactions execute mutually exclusively. To achieve this, Bonakdarpour et al. propose a three-layered implementation architecture: the bottom layer consists of distributed components, the middle layer consists of a number of interaction execution engines, each responsible for executing its own subset of all interactions, and the top layer resolves potential conflicts. In terms of this thesis and the Hybrid Approach, the bottom layer represents workers, while the middle layer roughly represents a multiplication expression of constraint automata. Importantly, however, Bonakdarpour et al. aim for a finer distribution granularity than I do, which requires them to handle conflicting interactions with their third layer. I avoid this problem in the Hybrid Approach, by putting constraint automata with "conflicting transitions" in the same subset at compile-time, thereby effectively serializing those transitions at run-time; for performance reasons, I prefer firing such transitions sequentially over adding an algorithm for conflict resolution.

Nonassociativity

G-multiplication satisfies associativity, whereas l-multiplication does not, as already shown in Figures 5.2 and 5.3. So far, I worked around this limitation by formulating the premises in Theorems 7, 9 and 10 in terms of carefully ordered lists of constraint automata. Although this works fine in theory, the need for such lists has a problematic consequence in practice: because $\odot$ models $(\cdot)$ (i.e., the cheap consensus algorithm, with only local communication between neighbors), the order of the constraint automata in the list at compile-time essentially fixes the order in which their protocol units may communicate with each other to reach consensus at run-time. In other words, because $(\cdot)$ actually l-multiplies constraint automata, and because l-multiplication does not satisfy associativity, $(\cdot)$ must ensure that its run-time multiplication "abides by compile-time parentheses". For instance, suppose that the third step in the Hybrid Approach yields three medium automata $\boldsymbol{b}_1$, $\boldsymbol{b}_2$, and $\boldsymbol{b}_3$. Moreover, suppose that $\boldsymbol{b}_1 \odot (\boldsymbol{b}_2 \odot \boldsymbol{b}_3) \simeq \boldsymbol{b}_1 \otimes \boldsymbol{b}_2 \otimes \boldsymbol{b}_3$ by Theorem 10, whereas $(\boldsymbol{b}_1 \odot \boldsymbol{b}_2) \odot \boldsymbol{b}_3 \not\simeq$

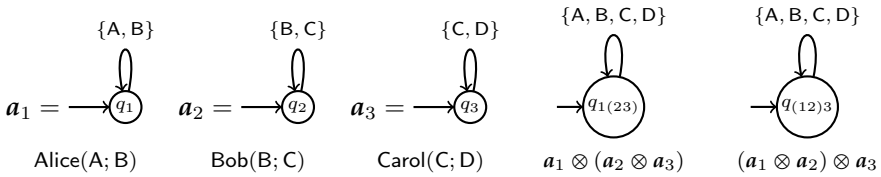


Figure 5.4: Two g-products of Alice(A; B), Bob(B; C), and Carol(C; D), without data constraints for simplicity. States $q_{1(23)}$ and $q_{(12)3}$ abbreviate $(q_1, (q_2, q_3))$ and $((q_1, q_2), q_3)$.

$\boldsymbol{b}_1 \otimes \boldsymbol{b}_2 \otimes \boldsymbol{b}_3$ by nonassociativity. In that case, instead of denoting the protocol subprogram subsequently generated in the fourth step by $\mathtt{b}_1$ `(·)` $\mathtt{b}_2$ `(·)` $\mathtt{b}_3$ (as suggested by the notation in Figure 5.1), perhaps I should denote this subprogram more precisely by $\mathtt{b}_1$ `(·)` $\left[\mathtt{b}_2 \texttt{ (·) } \mathtt{b}_3\right]$. After all, this notation explicitly shows that the protocol units defined by $\mathtt{b}_2$ and $\mathtt{b}_3$ must always communicate first with each other—and reach a local consensus (i.e., multiply $\boldsymbol{b}_2$ and $\boldsymbol{b}_3$)—before any of them can communicate with the protocol unit defined by $\mathtt{b}_1$. Generally, such fixed communication orders deteriorate performance.

Interestingly, *strictly speaking*, one may apply the same reasoning to `(x)`, even though $\otimes$ satisfies associativity. First, one may argue that although I do not write parentheses in $\boldsymbol{b}_1 \otimes \boldsymbol{b}_2 \otimes \boldsymbol{b}_3$ out of notational convenience (and because `(x)`'s associativity and commutativity make the placement of parentheses immaterial under behavioral congruence), *strictly speaking*, because $\otimes$ takes two multiplicands by Definition 29, that expression nevertheless has parentheses somewhere. Subsequently, one may argue that *strictly speaking*, I should denote the corresponding protocol subprogram either by $\mathtt{b}_1$ `(x)` $\left[\mathtt{b}_2 \texttt{ (x) } \mathtt{b}_3\right]$ or by $\left[\mathtt{b}_1 \texttt{ (x) } \mathtt{b}_2\right]$ `(x)` $\mathtt{b}_3$. Finally, one may conclude that *strictly speaking*, also the Distributed Approach fixes a communication order at compile-time, which protocol units should abide by at run-time. *Intuitively speaking*, however, such strictness makes little sense: $\otimes$ satisfies associativity, therefore the order of applying multiplications on constraint automata does not matter, therefore the order in which protocol units in the Distributed Approach communicate with each other does not matter.

The previous mismatch results from the lack in formal precision about how the protocol units defined by $\mathtt{b}_1$ `(x)` $\mathtt{b}_2$ `(x)` $\mathtt{b}_3$ should behave. Under a *strict perspective*, protocol units must respect not only the behavior of $\boldsymbol{b}_1 \otimes \boldsymbol{b}_2 \otimes \boldsymbol{b}_3$ but also this expression's structure (i.e., the hidden placement of parentheses). For performance reasons, however, I prefer a *loose perspective*: as long as protocol units respect the behavior of an expression, whether or not their communication order respects the structure of the expression should not matter.

(Perhaps I seem to diverge from my original goal at this point, dwelling on associativity in the Distributed Approach instead of dealing with nonassociativity in the Hybrid Approach. Shortly, however, I use the technique presented in the former context to give a solution for the problem in the latter context.)

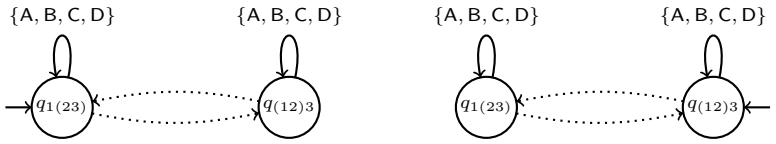


Figure 5.5: Behaviorally equivalent constraint automata to $\boldsymbol{a}_1 \otimes (\boldsymbol{a}_2 \otimes \boldsymbol{a}_3)$ and $(\boldsymbol{a}_1 \otimes \boldsymbol{a}_2) \otimes \boldsymbol{a}_3$ in Figure 5.4. Dotted arrows represent internal transitions.

To make the two different perspectives more concrete, recall the `AliceBob-CarolDave` example in Figure 4.7. As before, I anthropomorphize the protocol units in this example as Alice, Bob, and Carol (Dave plays no role here). Figure 5.4 shows two g-products of $\mathsf{Alice}(\mathsf{A};\mathsf{B})$, $\mathsf{Bob}(\mathsf{B};\mathsf{C})$, and $\mathsf{Carol}(\mathsf{C};\mathsf{D})$. These g-products differ primarily in the structure of their states, which reflects the placement of parentheses in their expressions. For Alice, Bob, and Carol to behave correctly under the strict perspective, depending on the placement of parentheses at compile-time, they must behave either as the right-associative or as the left-associative g-product in Figure 5.4 at run-time. The loose perspective, in contrast, allows Alice, Bob, and Carol to behave as *any* behaviorally equivalent—not necessarily behaviorally congruent—constraint automaton at run-time. For instance, they may behave as the constraint automata in Figure 5.5. In that case, whenever Alice, Bob, and Carol fire an internal transition (cf. silent transitions in process calculi), they effectively change their communication order: in $q_{1(23)}$, Bob and Carol must go first, while in $q_{(12)3}$, Alice and Bob must go first. In other words, if Alice started communicating with Bob to effectuate their previous instance of interaction, but if now Bob (instead of Alice) starts communicating with Carol (instead of Bob) to effectuate their next instance of interaction, somewhere in between those two instances, Alice, Bob, and Carol must have fired a silent transition from $q_{(12)3}$ to $q_{1(23)}$.

I can extend Figure 5.5 by taking not only associativity into account but also commutativity. Doing so yields the behaviorally equivalent "huge" automaton in Figure 5.6. One can easily check that, under this automaton, Alice, Bob, and Carol can freely switch between all possible communication orders at run-time. As such, this huge automaton provides a formal justification for why, in the Distributed Approach, Alice, Bob, and Carol may dynamically change their communication order. Next, I generalize this argument with a construction of huge automata for arbitrary $\otimes$-expressions.

Let $=_{\mathrm{AC}}$ denote the smallest equivalence relation induced by the following rules (i.e., add also rules for reflexivity, symmetry, and transitivity):

$$\frac{\boldsymbol{a}_1, \boldsymbol{a}_2 \in \mathbb{A}\mathrm{UTOM}}{\boldsymbol{a}_1 \otimes \boldsymbol{a}_2 =_{\mathrm{AC}} \boldsymbol{a}_2 \otimes \boldsymbol{a}_1} \qquad \frac{\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3 \in \mathbb{A}\mathrm{UTOM}}{\boldsymbol{a}_1 \otimes (\boldsymbol{a}_2 \otimes \boldsymbol{a}_3) =_{\mathrm{AC}} (\boldsymbol{a}_1 \otimes \boldsymbol{a}_2) \otimes \boldsymbol{a}_3}$$

Essentially, $=_{\mathrm{AC}}$ denotes equality up-to associativity and commutativity. Given this equality, let $\mathbb{A}\mathrm{UTOM}/=_{\mathrm{AC}}$ denote the quotient set of $\mathbb{A}\mathrm{UTOM}$ under $=_{\mathrm{AC}}$. Because $\otimes$ satisfies associativity and commutativity up-to behavioral congruence, every equivalence class in $\mathbb{A}\mathrm{UTOM}/=_{\mathrm{AC}}$ contains only behaviorally con-

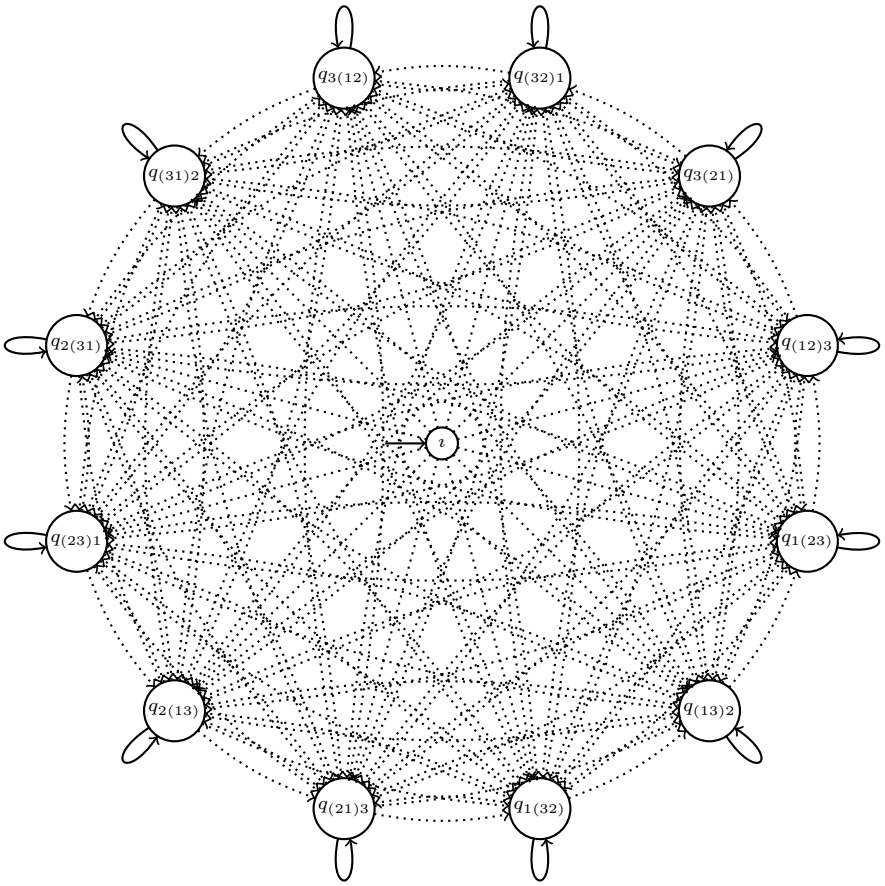


Figure 5.6: Huge automaton for Alice(A; B), Bob(B; C), and Carol(C; D), without data constraints for simplicity. Dotted arrows represent internal transitions; continuous arrows represent {A, B, C, D}-labeled transitions.

gruent constraint automata. Let $\boldsymbol{a}$ denote a constraint automaton, and let $A \in \mathbb{A}\text{UTOM}/=_{\text{AC}}$ denote its equivalence class. To construct a huge automaton for $\boldsymbol{a}$, first, I take the union of the constraint automata in A. This means that I take the union of their state spaces and transition relations and, because all constraint automata in A have the same ports, memory cells, and initial configuration, I simply copy those elements from any one of them into the huge automaton under construction. Second, I add a fresh initial state q^0 and connect this state to the (former) initial state of every $\boldsymbol{a} \in A$ with a $(\emptyset, \mathtt{K}(M))$-labeled transition. Such internal transitions have $\mathtt{K}(M)$ as their data constraint to ensure that their firing does not nondeterministically change the content of memory cells. Finally, I add a $(\emptyset, \mathtt{K}(M))$-labeled transition between all behaviorally congruent states, in both directions.

More formally, let $\mathcal{R}$ denote the largest set of relations such that $R \in \mathcal{R}$ implies both $\boldsymbol{a}_1 \preceq^{R} \boldsymbol{a}_2$ and $\boldsymbol{a}_2 \preceq^{R^{-1}} \boldsymbol{a}_1$ for some $\boldsymbol{a}_1, \boldsymbol{a}_2 \in A$ (cf. behavioral congruence in Definition 25).

$$\begin{aligned} Q \quad &= \{q^0\} \cup \bigcup\{\mathsf{Stat}(\boldsymbol{a}') \mid \boldsymbol{a}' \in A\} \\ \longrightarrow &= \bigcup\{\mathsf{Trans}(\boldsymbol{a}') \mid \boldsymbol{a}' \in A\} \\ &\quad \cup \{(q^0, \emptyset, \mathtt{K}(M), q') \mid \boldsymbol{a}' \in A \textbf{ and } \mathsf{init}(\boldsymbol{a}') = (q', \mu')\} \\ &\quad \cup \{(q, \emptyset, \mathtt{K}(M), q') \mid q \; R \; q' \textbf{ and } R \in \mathcal{R}\} \end{aligned}$$

I conjecture that on input of an arbitrary constraint automaton, "hugeification" so defined yields a behaviorally equivalent, yet behaviorally incongruent (because of its internal transitions), constraint automaton. After all, the added internal transitions by themselves have no effect on the accepted interaction language, and because these transitions connect only behaviorally congruent—hence, behaviorally equivalent by Theorem 1—states, a huge automaton has equivalent options for its behavior before and after firing these transitions.

The previous construction of huge automata essentially shows that protocol units in the Distributed Approach may dynamically change their communication order: under the loose perspective, protocol units collectively simulate not a constraint automaton that strictly reflects the placement of parentheses but a huge, behaviorally equivalent one. As such, hugeification—and in particular the definition of $=_{\mathrm{AC}}$—makes the importance of $\otimes$'s associativity formally precise: it (together with commutativity) characterizes sets of behaviorally equivalent constraint automata, each of which reflects a different communication order, between which protocol units may freely switch at run-time. Of course, no compiler should ever actually construct huge automata; I use hugeification only as a means for formal reasoning.

L-multiplication, in contrast to g-multiplication, does not satisfy associativity. Hence, I cannot directly apply the previous hugeification technique in the Hybrid Approach. Instead, I first need to establish under which conditions—if any—$\odot$ does satisfy associativity. I present such conditions in three lemmas. The first lemma states that if a constraint automaton $\boldsymbol{a}_2$ has enslaved a constraint automaton $\boldsymbol{a}_1$, neither of which depends on a constraint automaton $\boldsymbol{a}_3$, l-multiplication satisfies associativity.

Lemma 13.

$$\begin{bmatrix} \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3), (\boldsymbol{a}_1 \odot \boldsymbol{a}_2) \odot \boldsymbol{a}_3 \in \mathbb{A}\mathrm{UTOM} \\ \textbf{and } \boldsymbol{a}_1, \boldsymbol{a}_2 \asymp \boldsymbol{a}_3 \textbf{ and } \boldsymbol{a}_1 \bowtie \boldsymbol{a}_2 \end{bmatrix}$$
$$\textbf{implies } \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3) \simeq (\boldsymbol{a}_1 \odot \boldsymbol{a}_2) \odot$$

The second and the third lemma do not state associativity per se, but they state important properties otherwise implied by associativity (together with commutativity). The second lemma states that if both $\boldsymbol{a}_2$ and $\boldsymbol{a}_3$ have conditionally enslaved $\boldsymbol{a}_1$, while both $\boldsymbol{a}_1$ and $\boldsymbol{a}_3$ have conditionally enslaved $\boldsymbol{a}_2$ (i.e., $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ have enslaved each other), I can "swap" $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$. The third lemma has the same consequence but a different premise, namely that $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, and $\boldsymbol{a}_3$ have no dependencies between each other.

$$
\begin{aligned}
& \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_3 \odot (\boldsymbol{a}_4 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_6 \odot (\boldsymbol{a}_7 \odot \boldsymbol{a}_8)))))) \\
\simeq\ & a_1 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_3 \odot (\boldsymbol{a}_4 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_7 \odot (\boldsymbol{a}_6 \odot a_8)))))) \\
\simeq\ & a_1 \odot (\boldsymbol{a}_3 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_4 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_7 \odot (\boldsymbol{a}_6 \odot a_8)))))) \\
\simeq\ & a_1 \odot (a_3 \odot (\boldsymbol{a}_4 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_7 \odot (\boldsymbol{a}_6 \odot a_8)))))) \\
\simeq\ & a_1 \odot (a_3 \odot (a_4 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_7 \odot (\boldsymbol{a}_6 \odot a_8)))))) \\
\simeq\ & \boldsymbol{a}_3 \odot (\boldsymbol{a}_1 \odot (a_4 \odot (a_5 \odot (a_2 \odot (a_7 \odot (a_6 \odot a_8)))))) \\
\simeq\ & a_3 \odot (\boldsymbol{a}_4 \odot (\boldsymbol{a}_1 \odot (a_5 \odot (a_2 \odot (a_7 \odot (a_6 \odot a_8)))))) \\
\simeq\ & a_3 \odot (a_4 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_1 \odot (a_2 \odot (a_7 \odot (a_6 \odot a_8)))))) \\
\simeq\ & a_3 \odot (a_4 \odot (a_5 \odot ((\boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_7)) \odot (a_6 \odot a_8)))) \\
\simeq\ & a_3 \odot (a_4 \odot (a_5 \odot (\boldsymbol{a}_6 \odot ((\boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_7)) \odot a_8)))) \\
\simeq\ & a_3 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_4 \odot (a_6 \odot ((a_1 \odot (a_2 \odot a_7)) \odot a_8)))) \\
\simeq\ & a_3 \odot (a_5 \odot ((\boldsymbol{a}_4 \odot \boldsymbol{a}_6) \odot ((a_1 \odot (a_2 \odot a_7)) \odot a_8))) \\
\simeq\ & a_3 \odot (a_5 \odot ((\boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_7)) \odot ((\boldsymbol{a}_4 \odot \boldsymbol{a}_6) \odot a_8))) \\
\simeq\ & a_3 \odot (a_5 \odot (\boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_7 \odot ((a_4 \odot a_6) \odot a_8)))))
\end{aligned}
$$

$\boldsymbol{a}_7$ acquires $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$

$\boldsymbol{a}_6$ acquires $\boldsymbol{a}_4$

$\boldsymbol{a}_7$ releases $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$

Figure 5.7: Example reordering of a $\odot$-expression over slaves $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, $\boldsymbol{a}_3$, $\boldsymbol{a}_4$, and $\boldsymbol{a}_5$ and masters $\boldsymbol{a}_6$, $\boldsymbol{a}_7$, and $\boldsymbol{a}_8$

Lemma 14.

$$
\begin{bmatrix} \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3), \boldsymbol{a}_2 \odot (\boldsymbol{a}_1 \odot \boldsymbol{a}_3) \in \mathbb{AUTOM} \\ \textbf{and}\ \ \boldsymbol{a}_1 \bowtie \boldsymbol{a}_2, \boldsymbol{a}_3\ \ \textbf{and}\ \ \boldsymbol{a}_2 \bowtie \boldsymbol{a}_1, \boldsymbol{a}_3 \end{bmatrix}
$$
$$
\textbf{implies}\ \ \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3) \simeq \boldsymbol{a}_2 \odot (\boldsymbol{a}_1 \odot \boldsymbol{a}_3)
$$

Lemma 15.

$$
\begin{bmatrix} \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3), \boldsymbol{a}_2 \odot (\boldsymbol{a}_1 \odot \boldsymbol{a}_3) \in \mathbb{AUTOM} \\ \textbf{and}\ \ \boldsymbol{a}_1 \asymp \boldsymbol{a}_2, \boldsymbol{a}_3\ \ \textbf{and}\ \ \boldsymbol{a}_2 \asymp \boldsymbol{a}_1, \boldsymbol{a}_3 \end{bmatrix}
$$
$$
\textbf{implies}\ \ \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3) \simeq \boldsymbol{a}_2 \odot (\boldsymbol{a}_1 \odot \boldsymbol{a}_3)
$$

Each of these three lemmas has an intuitive meaning in terms of the masters/slaves interpretation of constraint automata in the premise in Theorem 9. Lemma 14 means that I can "move" every slave "rightward" until it neighbors the leftmost master. Dually, Lemma 15 means that I can move every master "leftward" until it neighbors the rightmost slave. Finally, Lemma 13 means that I can "group" any number of rightmost slaves and any number of leftmost masters together (i.e., put parentheses around them). The actual proofs of these results go by induction on the number of slaves and the number of masters. Combined, for a given master $\boldsymbol{a}_j$, I can use these lemmas to move $\boldsymbol{a}_j$ leftward, move all $\boldsymbol{a}_j$'s slaves rightward, and finally group $\boldsymbol{a}_j$ and its slaves such that, within this group, $\boldsymbol{a}_j$ constitutes a most deeply nested l-multiplication (several of which may exist). Figure 5.7 exemplifies such reordering.

Every reordering of a $\odot$-expression as in the premise in Theorem 9 transforms that expression into another expression. By Lemmas 13, 14, and 15, the l-products to which these $\odot$-expressions evaluate live in the same equivalence class (under behavioral congruence). Similarly, $\otimes$-expressions up-to associativity live in the same equivalence class (under behavioral congruence). Thus, by replacing the rule for associativity in the definition of $=_{\mathrm{AC}}$ with a rule for *conditional associativity* as stated in Lemmas 13, 14, and 15, I can construct a new equivalence relation $=_{\mathrm{cAC}}$, take the quotient $\mathbb{A}\mathrm{UTOM}/=_{\mathrm{cAC}}$, and hugeify in the same way as before. Every added internal transition in a huge automaton so constructed models a reordering. Because constraint automata in the same equivalence class under $=_{\mathrm{cAC}}$ have congruent behavior—hence, equivalent behavior by Theorem 1—the same argument for the correctness of hugeification applies as before.

The previous construction of huge automata essentially shows that only protocol units *for masters* in the Hybrid Approach may start communicating with their slaves at run-time. Moreover, by Definition 35 of slavery, none of these slaves need to communicate with protocol units other than their masters (i.e., each of their transitions that involves a shared port with a master involves only shared ports with that master). It does not matter which particular master among all masters goes first; expression reordering can occur for any master at any time. In fact, even multiple masters (with disjoint sets of slaves) may start communicating simultaneously. In that case, the corresponding expression just has multiple most-deeply nested multiplications (e.g., $(\boldsymbol{a}_1 \odot \boldsymbol{a}_2) \odot (\boldsymbol{a}_3 \odot (\boldsymbol{a}_4 \odot \boldsymbol{a}_5))$). Figures 5.8 and 5.9 show simplified event-handlers for a protocol unit that simulates a master/slave (cf. Figures 4.5 and 4.6). I do not intend these figures to convey real "algorithms"; they serve just as a stylized description of what event-handling roughly entails in the Hybrid Approach.

Nonassociative parallel composition operators (such as $\odot$) occur also in the literature on concurrency theory, where authors usually consider such operators defective. For instance, Vrancken set out to improve an earlier version of the Algebra of Communicating Processes (with the empty process), because the merge operator in that version "turned out not associative" [Vra97]; Baeten and Van Glabbeek acknowledge that "this problem [a nonassociative merge operator] was remedied" by Vrancken [BvG87]. In the context of timed automata with shared variables and action synchronization, Berendsen and Vaandrager point out that "the approach [to support shared variables and action synchronization] in [6] [sic] is flawed since parallel composition is not associative" [BV08]. Berendsen and Vaandrager also states that "commutativity and associativity are highly desirable properties for parallel composition operators" [BV08]. Anantharaman et al., in turn, consider a process algebra with a nonassociative synchronous composition operator, but they subsequently characterize a class of processes for which this operator actually exhibits associativity and work only with processes from that class [ACH05]. Segala discusses problems of defining a parallel composition operator for general probabilistic automata, symptomized by nonassociativity [Seg95]. Finally, Klin and Sassone notice that parallel composition in stochastic π-calculus generally fails

Input: a port p on which an event occurred, a *context* $P^{\text{ctxt}} \subseteq P^{\text{in}} \cup P^{\text{out}}$ of global boundary ports with a pending I/O operation, and the current local state q_i of $\boldsymbol{b}_i$
Output: q_i' holds the next local state of $\boldsymbol{b}_i$.
Effect: either, through the firing of enabled local transitions (including a local transition of $\boldsymbol{b}_i$), an enabled global transition fires (if the I/O operations pending on the ports in P^{ctxt} satisfy that transition's label), or all global transitions are disabled (otherwise).

1. Wake up, and assign q_i to q_i'.
2. Assign $\emptyset$ to Φ, a variable for a set of data constraints.
3. For all transitions $q_i \xrightarrow{P_i,\phi_i}_i q_i'$, ordered nondeterministically:
   (a) If $p \notin P_i$, continue (i.e., skip to the next iteration).
   (b) If $P_i \cap (P^{\text{in}} \cup P^{\text{out}}) \not\subseteq P^{\text{ctxt}}$ (i.e., not all boundary ports involved in the current local transition have a pending I/O operation), continue.
   (c) Assign $\{\phi\}$ to Φ', a variable for a set of data constraints.
   (d) For all ports $p' \in P_i \setminus (P^{\text{in}} \cup P^{\text{out}})$:
      i. Send a message to the protocol unit that shares access to p' to ask which data constraints must hold for that unit to fire a transition involving p'.
      ii. Await an answer message Φ'' from that protocol unit.
      iii. Assign $\{\phi' \wedge \phi'' \mid \phi' \in \Phi' \textbf{ and } \phi'' \in \Phi''\}$ to Φ'.
   (e) Assign $\Phi \cup \Phi'$ to Φ.
4. For all data constraints $\phi \in \Phi$:
   (a) Compute a data assignment σ that respects the pending I/O operations and satisfies ϕ; continue if no such σ exists.
   (b) Distribute data among local ports and memory cells according to σ.
   (c) Send σ to all protocol units sent messages to in Step 3.
   (d) Compute a q' such that $q_i \xrightarrow{P_i,\phi_i} q'$ and $P_i \subseteq \mathrm{Dom}(\sigma)$ and $\sigma \models \phi_i$.
   (e) Assign q' to q_i', and abort the loop.
5. If the previous loop never made it to Step 4-e, send $\emptyset$ (i.e., the empty data assignment) to all protocol units sent messages to in Step 3.
6. Go dormant.

Figure 5.8: Simplified p-event-handler for a protocol unit that simulates a master medium automaton $\boldsymbol{b}_i = (Q_i, (P_i^{\text{all}}, P_i^{\text{in}}, P_i^{\text{out}}), M_i, \longrightarrow_i, (q_i^0, \mu_i^0))$ in the Hybrid Approach, where P^{in} and P^{out} denote the sets of global input and output ports

to exhibit associativity and investigate under which conditions it does [KS08]. However, as with $\odot$ in this thesis, nonassociativity does not always pose problems. For instance, in the context of reachability analysis, Yeh investigates a state space reduction technique for processes by adding distinguished actions for suspending and resuming processes; in the resulting theory, parallel composition does not exhibit associativity [Yeh93]. As another example, Kuske &

Input: a port p on which an event occurred, a *context* $P^{\text{ctxt}} \subseteq P^{\text{in}} \cup P^{\text{out}}$ of global boundary ports with a pending I/O operation, and the current local state q_i of $\boldsymbol{b}_i$
Output: q_i' holds the next local state of $\boldsymbol{b}_i$.
Effect: either, through the firing of enabled local transitions (including a local transition of $\boldsymbol{b}_i$), an enabled global transition fires (if the I/O operations pending on the ports in P^{ctxt} satisfy that transition's label), or all global transitions are disabled (otherwise).

1. Wake up, and assign q_i to q_i'.
2. Assign $\emptyset$ to Φ, a variable for a set of data constraints.
3. For all transitions $q_i \xrightarrow{P_i,\phi_i}_i q_i'$, ordered nondeterministically:
   (a) If $p \notin P_i$, continue (i.e., skip to the next iteration).
   (b) If $P_i \cap (P^{\text{in}} \cup P^{\text{out}}) \not\subseteq P^{\text{ctxt}}$ (i.e., not all boundary ports involved in the current local transition have a pending I/O operation), continue.
   (c) Assign $\Phi \cup \{\phi\}$ to Φ.
4. Send an answer message Φ to the protocol unit from which the p-event originated.
5. Await a message with a data assignment σ.
6. If $\sigma \neq \emptyset$, distribute data among local ports and memory cells according to σ.
7. If $\sigma \neq \emptyset$, compute a q' such that $q_i \xrightarrow{P_i,\phi_i} q'$ and $P_i \subseteq \mathrm{Dom}(\sigma)$ and $\sigma \models \phi_i$.
8. If $\sigma \neq \emptyset$, assign q' to q_i'.
9. Go dormant.

Figure 5.9: Simplified p-event-handler for a protocol unit that simulates a slave medium automaton $\boldsymbol{b}_i = (Q_i, (P_i^{\text{all}}, P_i^{\text{in}}, P_i^{\text{out}}), M_i, \longrightarrow_i, (q_i^0, \mu_i^0))$ in the Hybrid Approach, where P^{in} and P^{out} denote the sets of global input and output ports

Meinecke introduce a nonassociative product operator on branching automata with costs [KM03]. Finally, the Orc orchestration language has three combinators to express parallel execution, two of which exhibit neither associativity nor commutativity [KQCM09].

L-multiplication as introduced in this chapter differs from previous nonassociative parallel composition operators in the sense that even though it fails to exhibit associativity in general, it exhibits associativity *in all relevant cases* (Lemmas 13–15). As such, l-multiplication's nonassociativity does not render its definition defective. On the contrary: l-multiplication's nonassociativity essentially reflects the inherent asymmetry between masters and their slaves and, consequently, constitutes a *feature*, not a *bug*.

5.2 Practice

(I have not yet submitted the material in this section for publication.)

Compiler

I extended Lykos with the ability to generate code under the Hybrid Approach, controllable through flag `PARTITION`. When raised, Lykos partitions the set of small automata as described in Section 5.1. Consequently, instead of generating only a single protocol subprogram (as in Chapter 4), Lykos generates multiple protocol subprograms, one for every previously computed medium automaton. Each of these protocol subprograms defines a protocol unit, and every such a protocol unit corresponds to either a master or a slave.

Recall from Chapter 4 that every port data structure at run-time has two users: a protocol unit and either another protocol unit or a worker unit. In the latter case, if a port data structure has a protocol unit and a worker unit as its users, I call this protocol unit "on the boundary". During compilation, to ease code generation, Lykos ensures that every protocol unit on the boundary corresponds to a master. The technique to do this consists, essentially, of adding for every small automaton that satisfies no-synchronization an extra Sync "before" each of its input ports and an extra Sync "after" each of its output ports (where "input" and "output" qualify ports from the protocol perspective). For instance, Lykos may replace Fifo$(p_1; p_4)$ with Sync$(p_1; p_2)$, Fifo$(p_2; p_3)$, and Sync$(p_3; p_4)$. First, observe that Fifo$(p_1; p_4)$ and the product of Sync$(p_1; p_2)$, Fifo$(p_2; p_3)$, and Sync$(p_3; p_4)$ have equivalent behavior. Indeed, Sync forms some kind of neutral element for multiplication; I come back to this point in Chapter 6. Thus, Lykos does not affect the original behavior of a set of small automata by adding Syncs as just described. Second, observe that because Fifo satisfies no-synchronization, it ends up in its own subset in the partition and, by itself, constitutes one medium automaton—a slave. At the same time, Sync$(p_1; p_2)$ and Sync$(p_3; p_4)$ share no ports and, thus, satisfy independence. Consequently, also each of these two Syncs ends up in its own subset in the partition and, by itself, constitutes a medium automaton—a master. By adding Syncs for all small automata that satisfy no-synchronization in this way, Lykos ensures that only protocol units for masters lie on the boundary.

As explained in Chapter 4, worker threads execute not only computation code but also interaction code, on behalf of protocol units for masters that lie on the boundary. In contrast to the situation in Chapter 4, however, every protocol unit for a master—including those on the boundary—additionally has its own thread. Usually, these protocol threads lie dormant. Only after a neighboring protocol unit (for a slave) has made a transition, a protocol thread awakes and starts a new round of event-handling. After all, if this neighbor has changed state during its previous transition, in its new current state, it may have different outgoing transitions, with different synchronization constraints, than before. Consequently, this neighbor may now agree to involve shared ports in transitions that it could not agree to before, in its previous state (where it had different outgoing transitions, with different synchronization constraints). All threads ensure that they execute event-handlers in mutually exclusive fashion, to avoid race conditions.

While every protocol unit for a master has its own thread, protocol units

for slaves do not have their own thread. Instead, threads that execute code (on behalf) of protocol units for masters also execute code on behalf of protocol units for slaves, in the same way that worker threads execute code on behalf of protocol units. In other words, the same thread interleaves the execution of Figure 5.8 (i.e., the protocol unit for a master) with the execution of Figure 5.9 (i.e., the protocol units for that master's slaves), without going to sleep, awaking other threads, or explicitly sending messages—the thread does all the work itself. As a consequence, protocol units for slaves essentially degenerate into purely passive entities—data structures—at run-time. For instance, the protocol unit for a Fifo effectively consists just of a variable (to hold the content of the buffer) and a lock (to guarantee mutual exclusion). Whenever a thread fires a transition of such a passive protocol unit for a slave, this thread also notifies all neighboring protocol units (for masters) of a possible state change, as already discussed above.

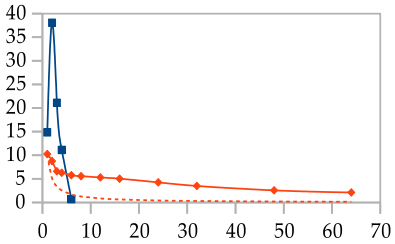
As another, minor optimization, Lykos carries out static analysis at compile-time in an attempt to safely *predict* whether a protocol unit for a master can reach consensus with its neighboring protocol units (for slaves) at run-time. This works as follows. Suppose that a master and its slave share a port p (and nobody else knows about p). Moreover, suppose that the slave has only one transition (q, P, ϕ, q') involving p (i.e., $p \in P$). Then, at compile-time, Lykos can establish two facts about the situation at run-time in which the master successfully fires a local transition involving p: (i) its slave necessarily has q as its current state, and (ii) the computed data assignment satisfies ϕ. Given those facts, by adding ϕ already to the data constraint of every transition involving p in the master at compile-time, communication and composition of data constraints at run-time becomes unnecessary. Such manipulation at compile-time does not strengthen the original data constraints too much, because the corresponding transitions can fire *only together* with transition (q, P, ϕ, q'), in the master's slave anyway. Moreover, if the compiler manipulates data constraints in this way already at compile-time, to see if the master and its slave can agree at run-time, a simple check for whether the slave has q as its current state suffices. This further reduces the overhead of the already cheap consensus algorithm necessary in the Hybrid Approach.

I skip code examples in this chapter, because the run-time library and the compiler-generated code remain largely the same as what I showed already in Chapter 4.

Experiments I: Protocols

I repeated the same experiments as in Chapter 4, generating code for members of families SyncK, FifoK, Merger, Router, LateAsyncMerger, EarlyAsyncMerger, OddFibonacci, and Chess with the `PARTITION`-flag raised, but otherwise under the same conditions as in Chapter 4. Figure 5.10 shows per-family performance charts, averaged over five runs. The solid lines represent the actual measurements; the dotted lines represent inverse-proportional growth with respect to $k = 1$. Recall from Chapter 4 that inverse-proportional growth forms

—same as in Figure 4.26—



SyncK

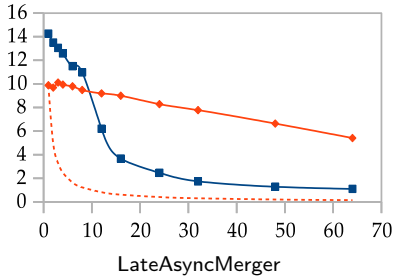
FifoK

—same as in Figure 4.26—

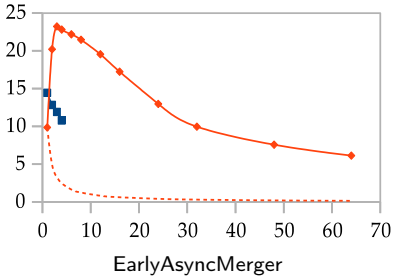
—same as in Figure 4.26—

Merger

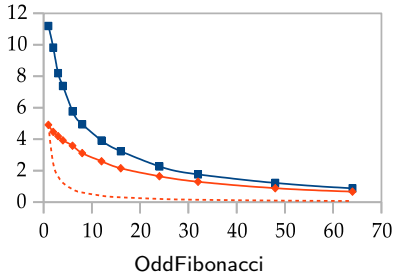
Router



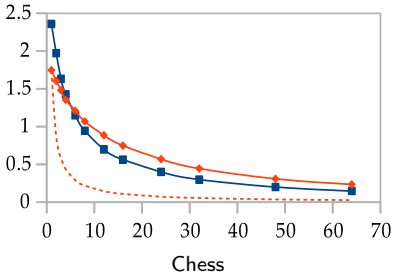
LateAsyncMerger



EarlyAsyncMerger



OddFibonacci



Chess

Figure 5.10: Performance (in number of completed rounds per four minutes) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

a necessary condition (but not necessarily a sufficient one) for good scalability. The red lines represent the new results; the blue lines represent the results from Chapter 4. For SyncK, Merger, and Router, Lykos generated exactly the same code as in Chapter 4. In these cases, the Hybrid Approach degenerates into the Centralized Approach.

Figure 5.11 shows per-family speedup charts corresponding to the measurements in Figure 5.10; the dotted lines represent equal performance. For FifoK, the previous scalability problems, both at compile-time and at run-time, have disappeared: at compile-time, Lykos (with the `PARTITION`-flag raised) succeeded in generating code for all values of k without exhausting its available resources, while at run-time, the performance of the generated code stays above the critical threshold of inverse-proportionality. For EarlyAsyncMerger, the same observations hold true. In these cases, thus, the Hybrid Approach indeed solves the previous scalability problems. For LateAsyncMerger and Chess, performance has also improved to greater and to lesser extent, indicating that the recovery of useful parallelism as achieved in the Hybrid Approach can pay off. But, code generated under the Centralized Approach for members of LateAsyncMerger and Chess with smaller values of k outperforms code generated under the Hybrid Approach for those same members. This indicates that parallelism becomes more important as the number of workers increases.

Interestingly, FifoK and EarlyAsyncMerger exemplify situations where code generated under the Hybrid Approach has, in fact, *lower* latency than code generated under the Centralized Approach. This may come as a surprise, as the Hybrid Approach requires a (cheap) consensus algorithm (which nevertheless inflicts overhead and thereby generally increases latency), whereas the Centralized Approach does not. Sometimes, thus, code generated under the Centralized Approach has *another* major source of overhead, which increases latency more dramatically than the Hybrid Approach's consensus algorithm does (e.g., for members of FifoK and EarlyAsyncMerger for certain values of k). This source of overhead consists of the number of transitions that a big automaton, as computed in the Centralized Approach, may have: whenever a big automaton has many more transitions than every medium automaton in the Hybrid Approach—this happens, for instance, if that number of transitions increases exponentially in k, as with FifoK—it may take much longer for that big automaton's protocol unit to find an enabled transition than for the protocol units for those medium automata. This observation, then, constitutes another point in favor of the Hybrid Approach.

Finally, for OddFibonacci, all code generated under the Centralized Approach actually outperforms all code generated under the Hybrid Approach. Here, one witnesses *overparallelization*, where a number of parallel threads implement an inherently sequential protocol specification. For all members of OddFibonacci, the computed partition consists of seven subsets (independent of the number of consumers), subsequently resulting in seven medium automata: three masters and four slaves (i.e., Fifos). However, as shown in the constraint automaton for the $\mathsf{OddFibonacci}_2$ protocol in Figure 3.25, this protocol has no real parallelism to exploit among its workers: either all workers

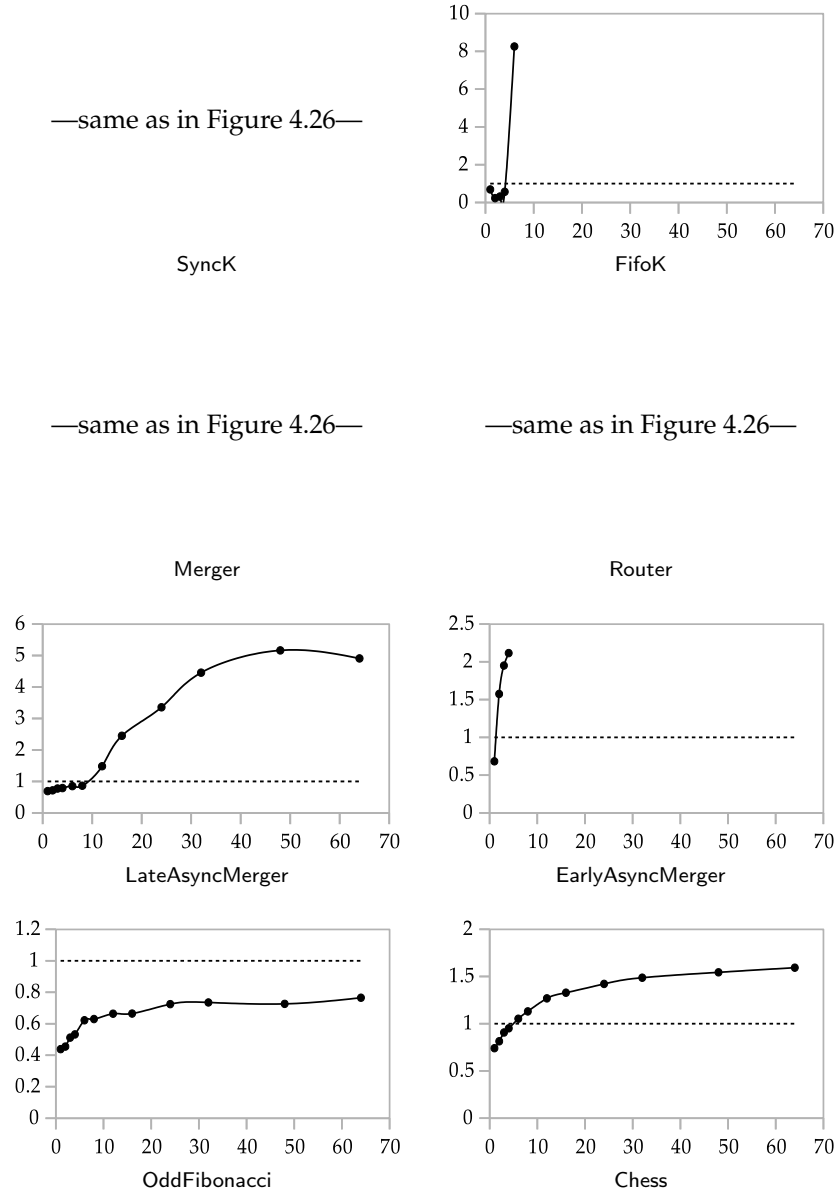


Figure 5.11: Speedup (relative to compiler-generated code in Chapter 4) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

synchronously complete their I/O operations (the left transition in Figure 3.25), or the producer completes an I/O operation just by itself (the right transition). Consequently, a parallel implementation of seven protocol units (three active ones and four passive ones) incurs the overhead that parallelism entails without gaining anything. As a result, such parallel implementations have poorer performance than sequential ones, as shown in Figures 5.10 and 5.11.

Despite their overparallelization, the seven protocol units for every OddFibonacci member exhibit "useful parallelism" according to its definition in Section 5.1, a concept that I used as a guideline for computing reasonable partitions. This suggests that to avoid overparallelization through more clever partitioning, I need to refine the definition of useful parallelism. After all, as the experiments with OddFibonacci members demonstrate, the current definition sometimes erroneously qualifies parallelism among protocol units as useful—and thereby essentially misguides the computation of reasonable partitions—while in practice, it leads to overparallelization.

To better explain the problem at hand, let a "sequence of dependent protocol units" start with one protocol unit, which depends on the next protocol unit, which, in turn, depends on the third protocol unit, and so on. If such a sequence of dependent protocol units starts and ends with the same protocol unit, and none of the intermediate protocol units lie on the boundary, the parallelism among those intermediate protocol units serves no real purpose. For instance (cf. OddFibonacci in Figures 3.26 and 3.27), suppose that the intermediate protocol units in a sequence of dependent protocol units correspond to a Fifo, followed by a Sync, followed by another Fifo. Also, suppose that the first/last protocol unit corresponds to a constraint automaton with a transition involving (at least) the input port of the first Fifo and the output port of the second Fifo. After firing this transition, the first/last protocol unit must wait for the intermediate protocol units to fire their transitions (i.e., to transport the new datum in the first buffer to the second buffer) before it can fire this transition again. The parallelism among those protocol units, therefore, has no real advantage—but, in contrast, negatively affects performance—and sequentializing this whole sequence of dependent protocol units seems the better option. In terms of the theory presented in this chapter, I can achieve such sequentialization by letting the subset for the first/last protocol unit in the sequence *gobble up* the subsets for the intermediate protocol units. Although I have a fair understanding of the theory involved in this proposal, the actual practical consequences remain unclear to me and require further investigation. In particular, one possible adverse side-effect that I foresee consists of the reintroduction of state space explosion (e.g., if the sequence of dependent protocol units in the previous example consists of 64 Fifos, merging their corresponding subsets in the partition eventually gives rise to a constraint automaton with 2^{64} states). I leave a thorough study of this topic for future work.

	NPB-FT	NPB-MG	NPB-CG	NPB-IS	NPB-BT	NPB-SP	NPB-LU
W	32	∞	∞	∞	22	34	31
A	128	∞	∞	∞	62	62	62
B	256	∞	∞	∞	100	100	100
C	512	∞	∞	∞	160	160	160

Figure 5.12: Maximum number of slaves in the NPB benchmarks

Experiments II: Programs

I repeated the same experiments as in Chapter 4, generating code for the NPB benchmarks with the `PARTITION`-flag raised, but otherwise under the same conditions as in Chapter 4. Using the Hybrid Approach, in contrast to the Centralized Approach as used in Chapter 4, Lykos succeeded in generating code for all values of k for all NPB kernel benchmarks, except NPB-MG, and for all but the last value of k for the three NPB application benchmarks and NPB-MG.

To facilitate a fair comparison of the performance of different implementations of NPB, the NPB documentation—available at the NASA website—defines five problem size classes, with predefined inputs, to run the benchmarks on: class S (small size, just for testing), class W (1990s workstation size), class A (1990s supercomputer size, larger than class W), class B (1990s supercomputer size, roughly four times larger than class A), and class C (1990s supercomputer size, roughly four times larger than class B). I ran every FOCAML-to-Java-compiled version of the NPB kernel benchmarks with inputs from class W, class A, class B, and class C, while I ran every FOCAML-to-Java-compiled version of the NPB application benchmarks with inputs only from class W and class A; the latter benchmarks took, in the slowest cases, already over an hour, which made further upscaling the problem size impractical. I ran these benchmarks for every value of $k \in \{2, 4, 8, 16, 32, 64\}$ for which Lykos succeeded in generating code, where k denotes the number of slaves, except if such a k exceeded the maximum number of slaves for a given benchmark/class combination in Figure 5.12 (these limits come from the Java implementation of NPB). As before, I used a machine with 24 cores (two Intel E5-2690V3 processors in two sockets), without Hyper-Threading and without Turbo Boost (i.e., with a static clock frequency).

Figures 5.13–5.20 show performance charts for the FOCAML-to-Java-compiled versions of the NPB kernel benchmarks (averaged over five runs), speed-up charts (with respect to their Java versions by Frumkin et al.), and charts about cache misses. The dotted red lines represent the MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks; the solid red lines represent the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions; the dotted black lines represents the Java versions by Frumkin et al.

The machine on which I performed my experiments allowed me to monitor

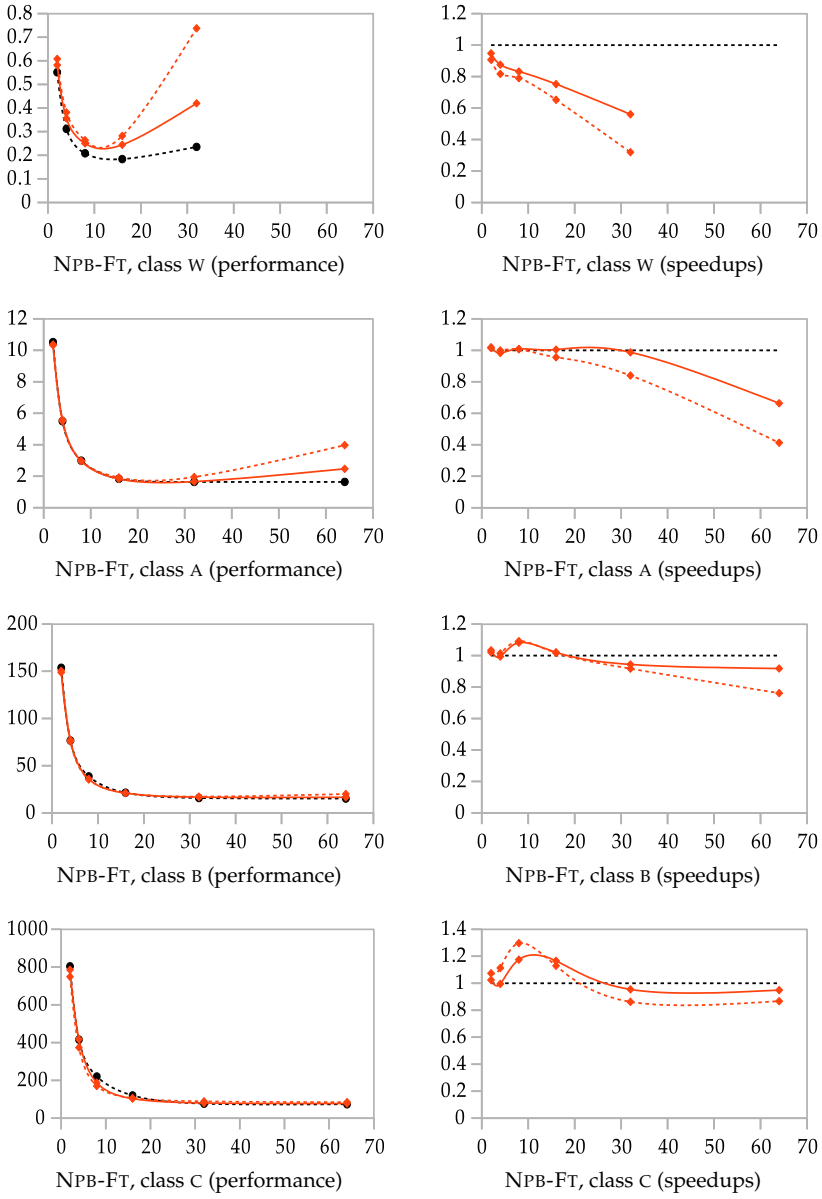


Figure 5.13: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

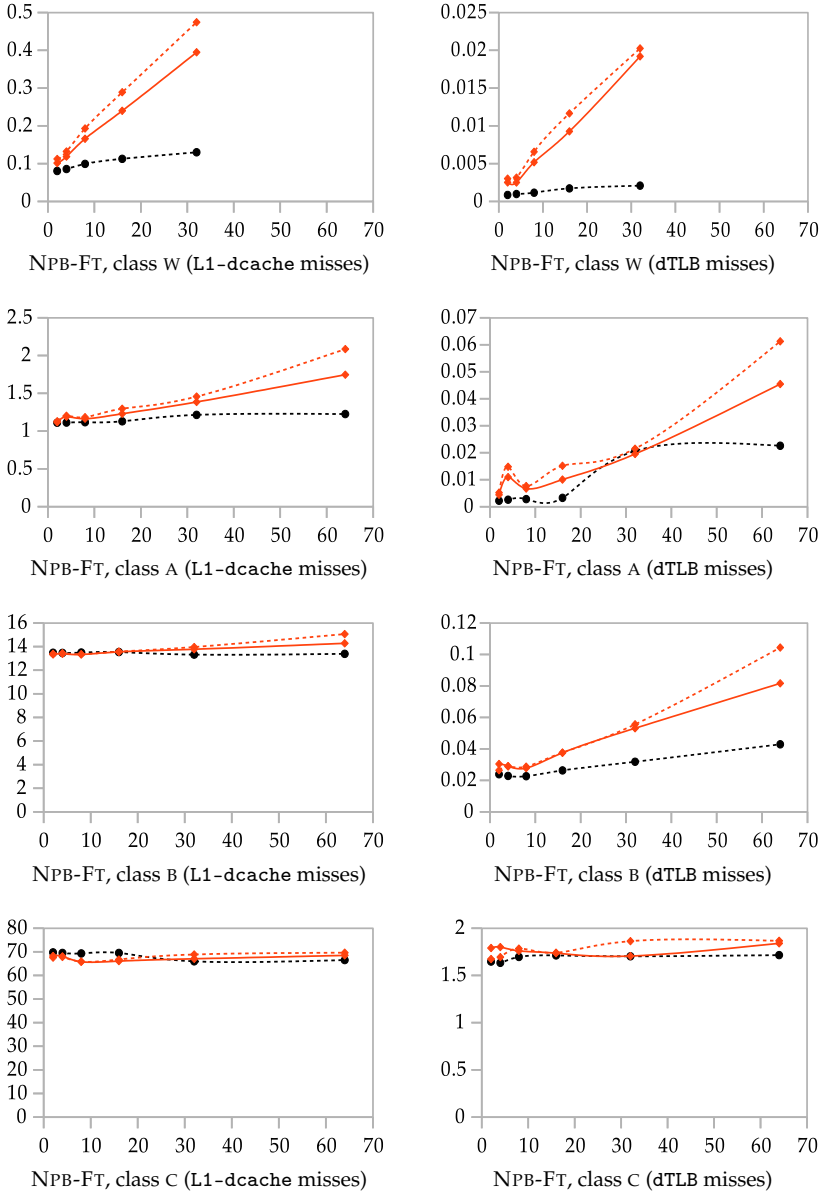


Figure 5.14: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

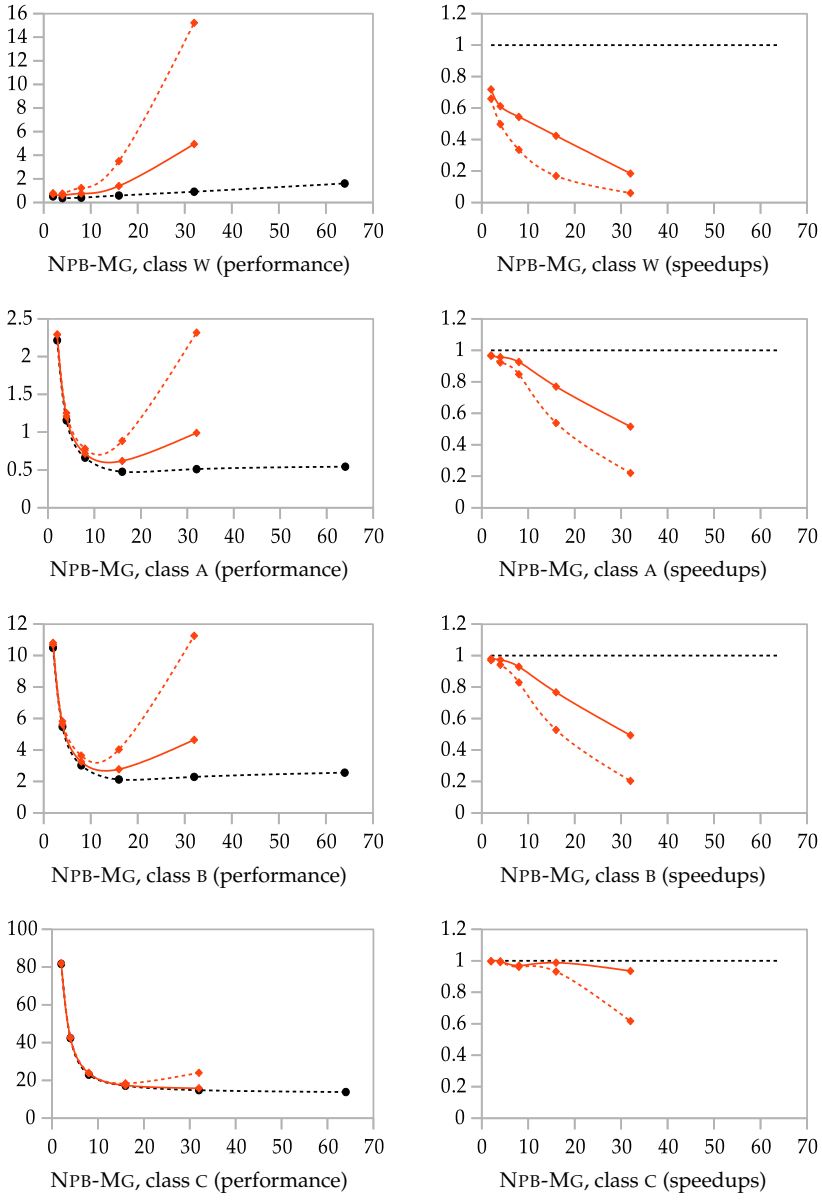


Figure 5.15: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

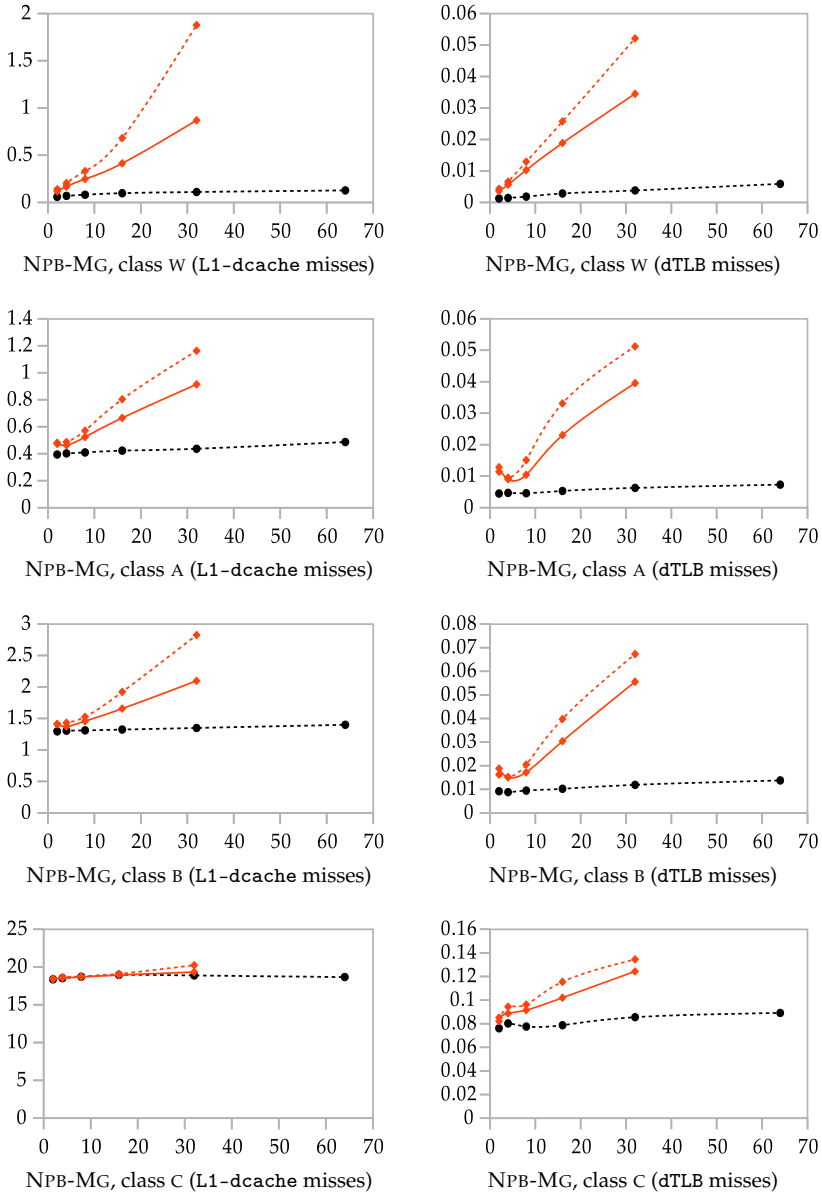


Figure 5.16: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

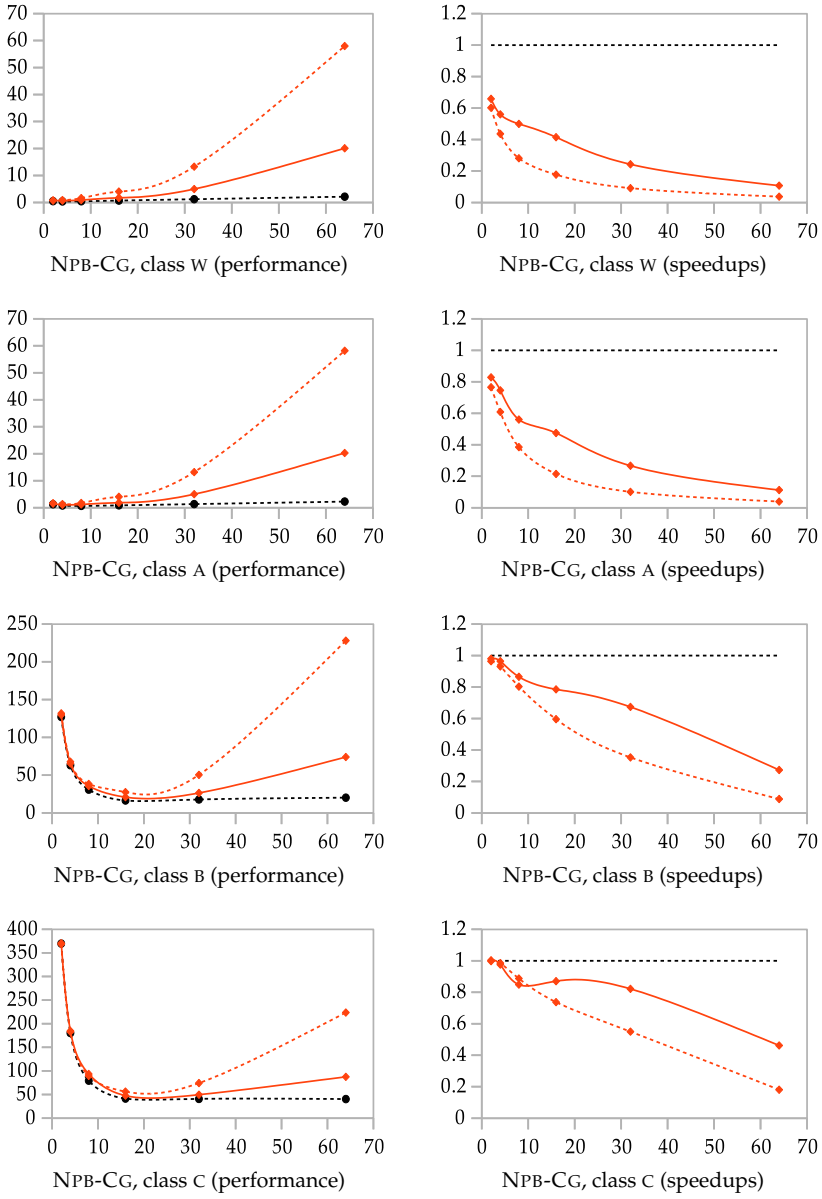


Figure 5.17: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

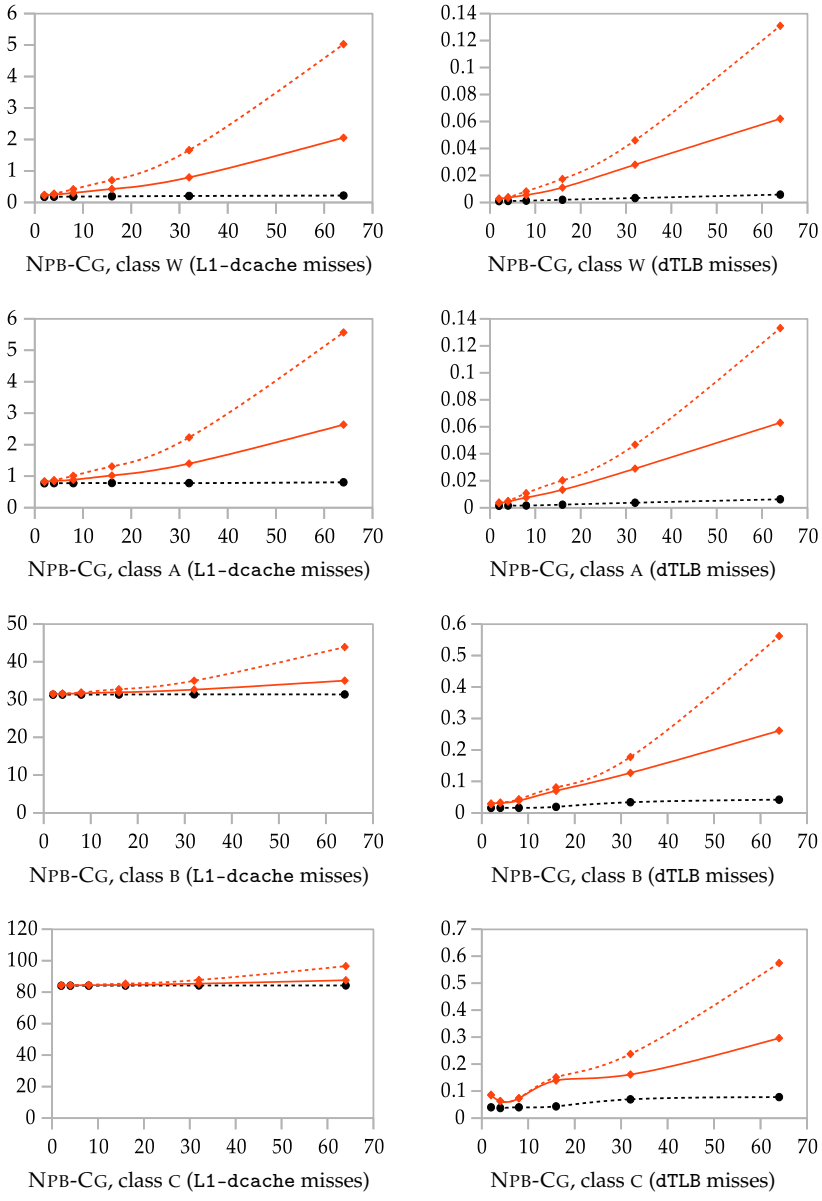


Figure 5.18: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

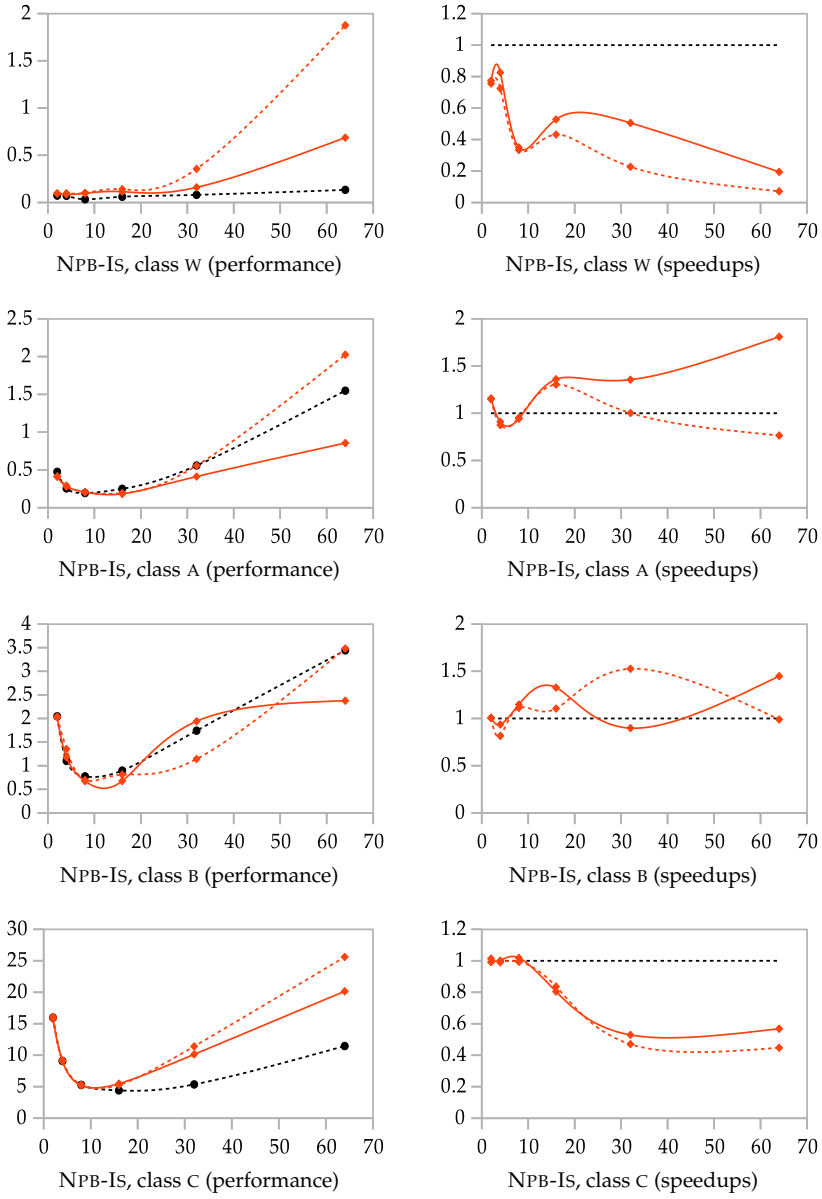


Figure 5.19: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

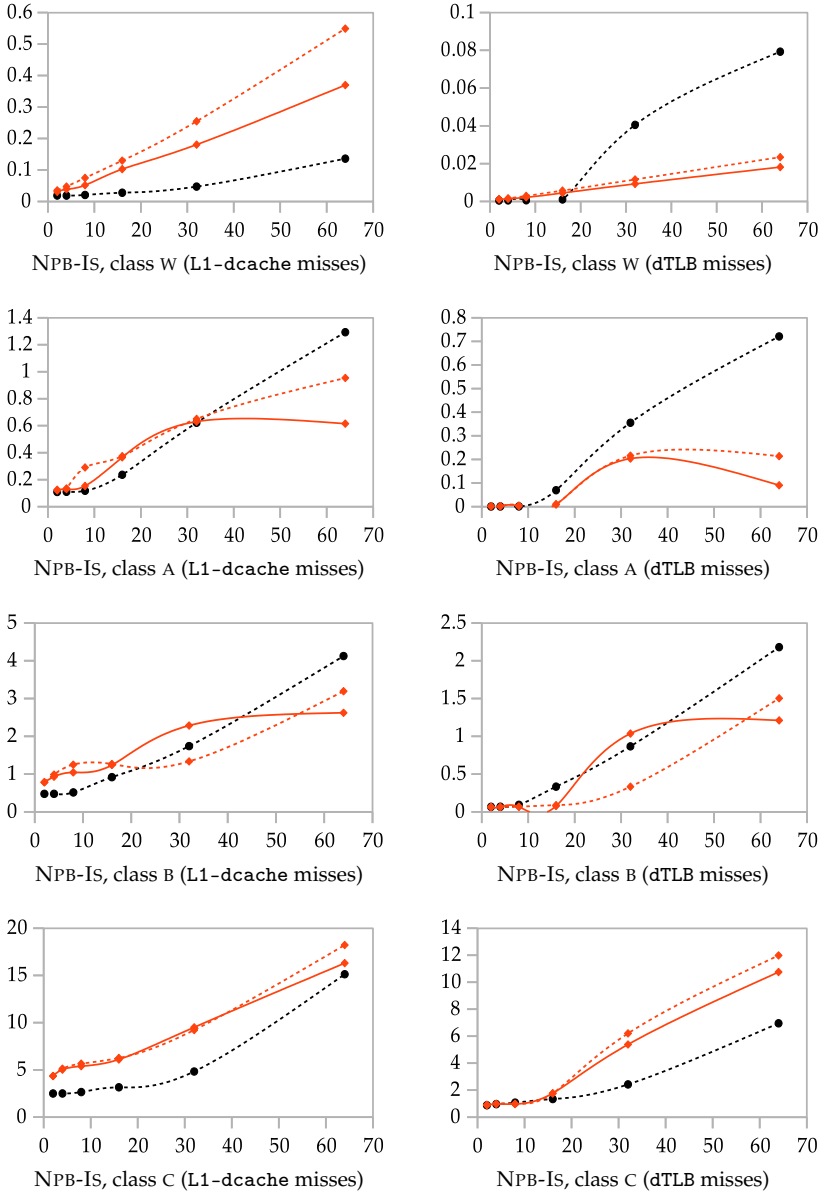


Figure 5.20: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

	cycles/miss	nanoseconds/miss	seconds/billion misses
`L1-dcache`	≥ 10	≥ 3.8	≥ 3.8
`dTLB`	7	2.3	2.3

Figure 5.21: Indication of costs involved in `L1-dcache` and `dTLB` misses on a 2.6 GHz machine with Intel i7 architecture [HP11b]

	NPB-FT	NPB-MG	NPB-CG	NPB-IS	NPB-BT	NPB-SP	NPB-LU
W	34	881	1155	20	1800	5600	2400
A	34	121	1155	20	1800	5600	2000
B	104	601	5775	20	1800	5600	2000
C	104	681	5775	20	1800	5600	2000

Figure 5.22: Number of times that a master dispatches work to its slaves and subsequently waits for their signal in the NPB benchmarks. To get the number of transitions fired by a MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled version with k slaves, compute $4kn$, where n comes from this figure; to get the number of transitions fired by a MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled version with k slaves, compute $(1 + 3k)n$.

two kinds of *cache misses*: `L1-dcache` misses, which occur when the L1 cache of the CPU does not contain a requested piece of data, and `dTLB` misses, which occur when the data TLB contains no entry for a provided virtual address to translate into a physical address. Importantly, I measured cache misses during the whole run of a program, from start to end, using `perf-stat`. These numbers, thus, include also cache misses incurred during a program's initialization and finalization. In contrast, the time measurements, for which I reused the original code by Frumkin et al., start only after initialization and end already before finalization (i.e., my cache miss measurements span a longer interval than my time measurements). Therefore, take my measurements on cache misses with a grain of salt and use them just as a rough indication of cache behavior. Even under this proviso, though, these measurements give meaningful insight into the performance of compiler-generated code, as explained shortly. Figure 5.21 shows a conservative estimation of the costs involved in cache misses.

I make the following main observations about my experimental results:

- Overall, the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks outperform their MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions (solid red lines versus dotted red lines). Recall from Chapter 3 that the latter versions impose an order in which the master sends signals to its slaves and vice versa, whereas the former versions do not impose such an order. As hinted at already in that chapter, not imposing an order indeed seems

to result in better performance.

- Despite the previous point, the Java versions of the NPB kernel benchmarks by Frumkin et al. outperform many of their FOCAML-to-Java-compiled versions by a substantial margin. This shows that the code generated by Lykos leaves room for further improvements.

- The numbers of cache misses seem a fair indicator of performance: fewer cache misses generally means better performance. This suggest that not only the number of machine instructions derived from compiler-generated code matters, but also its impact on memory and cache usage. For instance, the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks generally incur substantially fewer cache misses than their MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions.

  Cache misses alone do not completely determine performance, though; the number of machine instructions surely plays a role too (demonstrated more clearly in the previous subsection, where I reported on experiments with protocols in isolation). Consider, for instance, NPB-IS, class A. Although the MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled version for $k = 64$ incurs substantially fewer cache misses than the corresponding Java version, the latter version nevertheless substantially outperforms the former one.

- Going from class W to class A, from class A to class B, and from class B to class C, the speedup of the FOCAML-to-Java-compiled versions—or rather, their *slowdown*—generally improve: as the problem size increases, computation time progressively dominates total time, because interaction time stays nearly constant (i.e., the fraction of interaction time divided by computation time decreases as the problem size increases). As a measure for interaction time, Figure 5.22 shows the number of times that a master dispatches work to its slaves and subsequently waits for their signal. Increasing problem sizes, thus, work in favor of relatively slow implementations of protocol specifications due to "amortization of slowness" over a longer total time.

- The previous point applies to NPB-FT, NPB-MG, and NPB-CG but seems not to apply to NPB-IS, for which Figure 5.19 shows perhaps confusing experimental results, especially for $k = 64$: in class W and class C, the Java versions outperform the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions (dotted black lines versus solid red lines), but in class A and class B, the latter versions somewhat surprisingly outperform the former versions. I speculate that this has something to do with numbers of cache misses: as shown in Figure 5.20, for $k = 64$, in class W and class C, the former versions incur substantially fewer cache misses than the latter versions, but in class A and class B, the latter versions incur substantially fewer cache misses than the former versions.

To substantiate my speculation, as a first sanity check, I try to estimate the effect of cache misses on performance in the following *conservative* back-of-the-envelope calculation for $k = 64$ in class A. Using the numbers in Figure 5.21, the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled version spends *at least* 2.5 seconds more on cache misses than the Java version. By dividing this difference by the number of cores—a coarse estimation of how cache misses affect wall clock time—I get roughly a 100 milliseconds delay per core, about six times smaller than the difference in performance in Figure 5.19 (assuming a uniform distribution of the computational load over cores). Because I calculated conservatively (i.e., L1 cache misses cost substantially more than 10 cycles if also the L2 cache does not contain the required data; cache misses and computational load do not uniformly distribute over cores), these 100 milliseconds form a lower bound to the real delay per core caused by cache misses. If anything, this calculation shows that my speculation, that cache misses have a large enough impact to account for the observed difference in performance, seems at least *not un*reasonable.

Having passed the sanity check, next, I investigate the phenomenon at hand by studying the underlying reason for differences in cache misses. It turns out that this correlates with how the Java virtual machine manages the heap, the details of which I skip. Essentially, because the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled version has a different memory usage than the Java version, the Java virtual machine infers different sizes for the young/old generation portions of the heap for these two versions. These different sizes manifest in different cache behavior. To witness this, by explicitly setting the size of the young generation portion to the same value for both versions under study, not only their number of cache misses become similar *but also their performance*. This, then, serves as evidence for my previous speculation that their difference in number of cache misses causes the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions to outperform the Java versions in class A and class B for $k = 64$.

Figures 5.23–5.28 show performance charts for the FOCAML-to-Java-compiled versions of the NPB application benchmarks (averaged over five runs), speedup charts (with respect to their Java versions by Frumkin et al.), and charts about cache misses. The lines have the same meaning as in the previous charts for the NPB kernel benchmarks. Recall from Figure 5.12 that NPB-BT and NPB-LU do not support more than 22 and 31 slaves in class W. Therefore, I have no measurements beyond $k = 16$ in class W for those benchmarks.

Essentially, the same observations apply here as for the previous experimental results of the NPB kernel benchmarks. Different from the FOCAML-to-Java-compiled versions of the NPB kernel benchmarks, however, the FOCAML-to-Java-compiled versions of the NPB application benchmarks have similar performance as their Java versions—and in some cases even better. The previous point about increasing problem sizes applies here too, though. Finally,

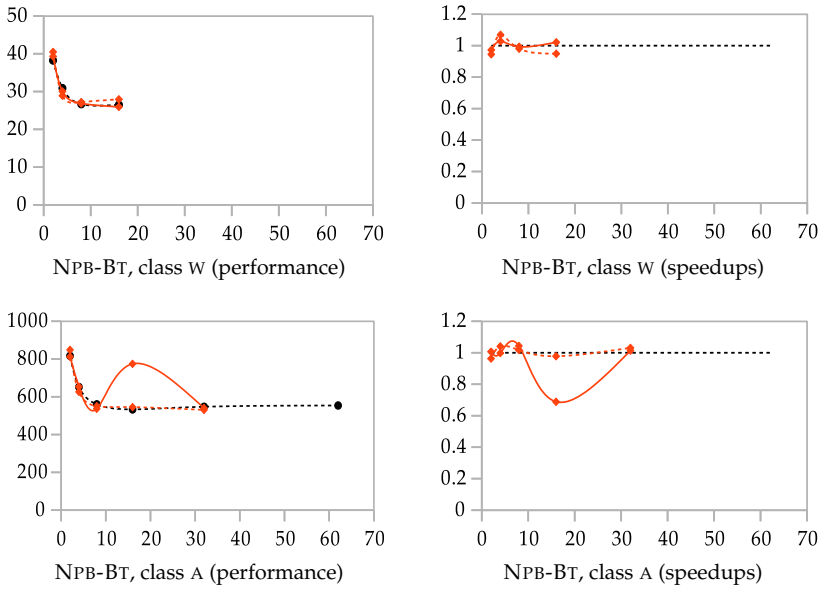


Figure 5.23: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

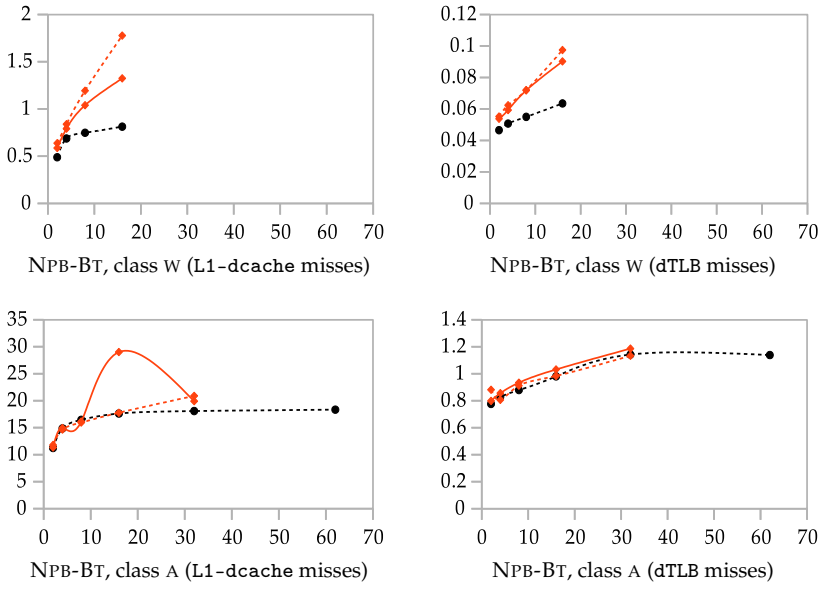


Figure 5.24: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

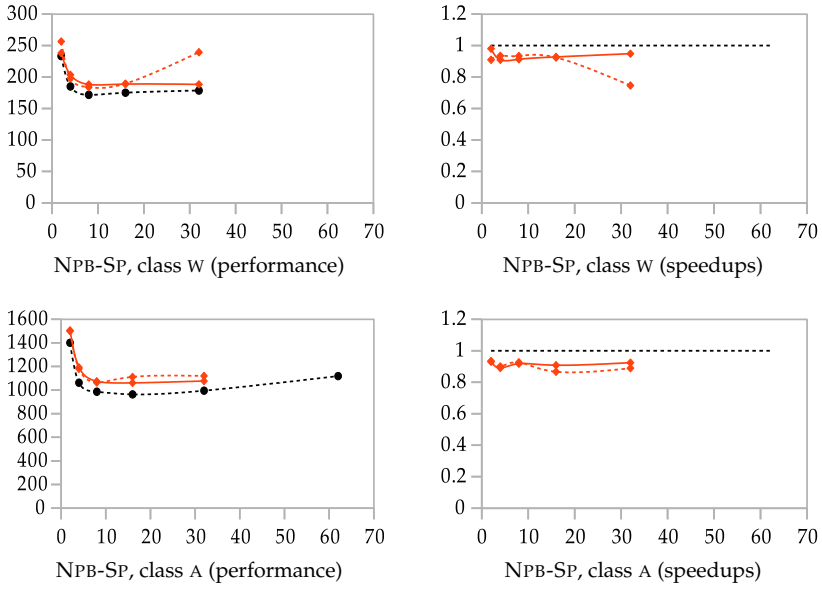


Figure 5.25: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

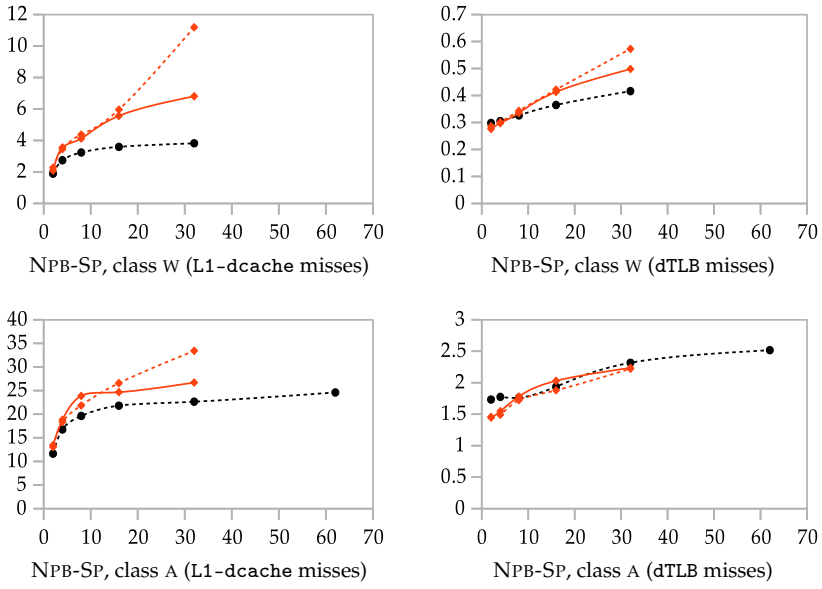


Figure 5.26: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

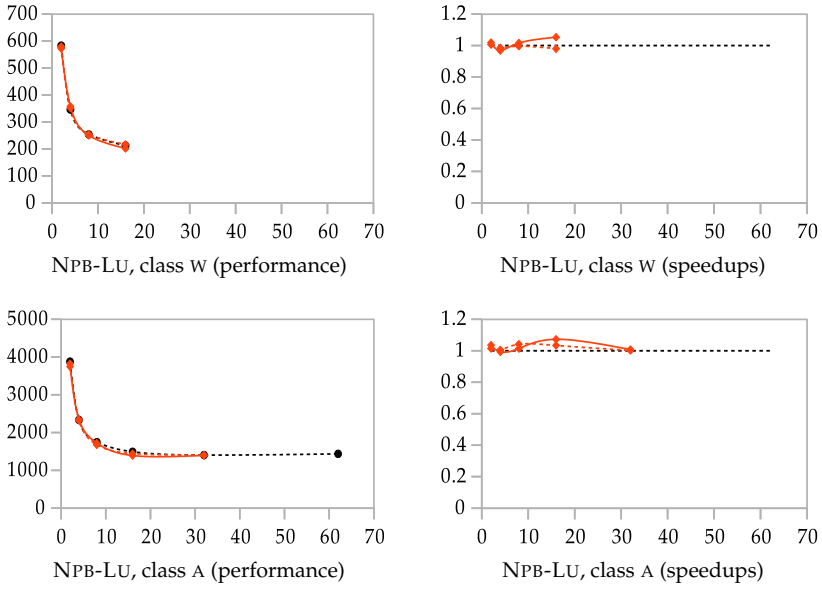


Figure 5.27: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

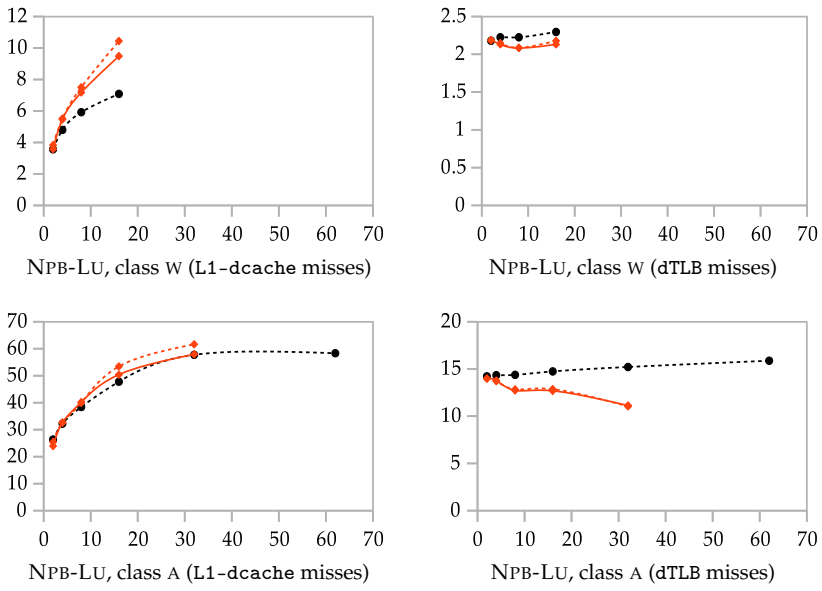


Figure 5.28: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

one anomalous run among five runs, which inexplicably took over three times longer to finish (i.e., half an hour versus 10 minutes), causes the spike for $k = 16$ in NPB-BT, class A, in Figure 5.24.

Their concrete results aside, this first round of experiments with the NPB benchmarks teaches an important lesson: evaluating the performance of compiler-generated code in the context of full programs yields new insights and requires more advanced analysis techniques (e.g., measuring and interpreting cache misses) than evaluating compiler-generated code in isolation, with empty producers and consumers. The experiments in this subsection, thus, nicely complement the experiments in the previous subsection.

Chapter 6

Improved Compilation II: Syntactic Subtraction

In Chapters 4 and 5, I presented a basic FOCAML compiler and reported on a technique for solving scalability problems of the Centralized Approach. Essentially, the Hybrid Approach presented in Chapter 5 tries to balance sequentiality *inside* protocol units with parallelism *among* those units. However, every individual protocol unit still executes purely sequential code and, as such, may form a bottleneck, potentially slowing down the entire program-in-execution.

In this chapter and the next, I present improvements to one of the more computationally costly aspects of protocol units' execution: the data constraint checks involved in determining whether a transition can fire and, in particular, the expensive constraint solver calls made during such checks. More precisely, in this chapter, I study a way of reducing the size of data constraints at compile-time to reduce the overhead of constraint solving at run-time. In Section 6.1, to motivate the need for this reduction, I first discuss a so far disregarded deficiency of the current compilation approach, related to the size of data constraints. Subsequently, I introduce an auxiliary operation on constraint automata, namely *normalization*, primarily to simplify subsequent technicalities. Finally, I introduce a new subtraction to carry out data constraint reduction and formally compare it to the old subtraction in Definition 30. In Section 6.2, I present an improved version of Lykos using this reduction technique for data constraints, including new experimental results on performance.

Although the improvement presented in this chapter eventually results in improved compiler-generated code, as in Chapter 5, I define this improvement at the higher level of constraint automata instead of at the lower level of GPL code. Not only does this facilitate more elegant formal reasoning about correctness (compared to reasoning directly about GPL code), but it also eases the automatic application of this improvement by a FOCAML compiler. Moreover, it makes this improvement independent of GPLs—Java in this thesis—so that the same optimization automatically applies to, for instance, generated C code.

6.1 Theory

(I have not yet published the material in this section.)

64 Syncs

Recall the Sync family defined in Figure 3.4. An undergraduate student of Computer Science may quite straightforwardly prove that members of Sync behave as a kind of algebraic identity of multiplication and subtraction, in the following sense. Let $\boldsymbol{a}$ range over the set of all constraint automata that have an input port p_2.

$$\mathsf{Behav}((\mathsf{Sync}(p_1;p_2)\otimes\boldsymbol{a})\ominus p_2) = \left\{ w' \;\middle|\; \begin{array}{l} w \in \mathsf{Behav}(\boldsymbol{a}) \\ \textbf{and } \left[w'(i) = w(i)|_{\mathrm{Dom}(w(i))\setminus\{p_2\}} \cup \{p_1 \mapsto w(i)(p_2)\} \textbf{ for all } i\right] \end{array} \right\}$$

In words, $(\mathsf{Sync}(p_1;p_2)\otimes\boldsymbol{a})\ominus p_2$ and $\boldsymbol{a}$ have language equivalent behavior modulo substitution of p_1 for p_2. Generally, one can "prefix" (i.e., multiply on its input ports) or "suffix" (i.e., multiply on its output ports) any number of Syncs to a constraint automaton without affecting—in the sense just described—that automaton's behavior. Given this property, it seems not unreasonable to assume that compiler-generated code for a single Sync has the same performance as a sequence of 64 Syncs. Slightly more formally, if $\sim$ means "has the same performance", one may expect:

$$\mathsf{Sync}(p_1;p_{65}) \sim (\mathsf{Sync}(p_1;p_2)\otimes\cdots\otimes\mathsf{Sync}(p_{64};p_{65}))\ominus p_2\ominus\cdots\ominus p_{64}$$

Sensible as this supposition may seem, as the experimental results in Chapter 4 show, code generated by Lykos violates this property: Sync_1 completes 27 million rounds in four minutes, whereas Sync_{64} completes only nine million rounds in the same amount of time (Figure 4.26). Indeed, although the performance of Sync_k stays above the critical threshold of inverse-proportionality, I can hardly claim that the compiler-generated code for Sync_k scales well in k. Thus, as I already stated before, inverse-proportionality forms a necessary condition for good scalability but not necessarily a sufficient one; I discuss inverse-proportionality for the other families of constraint automata with which I experimented so far in Section 6.2.

To better understand the phenomenon at hand, take another look at Definition 30 of $\ominus$. Whereas subtraction eliminates ports from synchronization constraints *syntactically*—effectively making those sets smaller—it removes ports from data constraints only *semantically*. Indeed, $\ominus$ does not reduce the size of data constraints (in terms of the number of data variables, data literals, and existential quantifications) but, in fact and in contrast, makes data constraints larger by enveloping them in existential quantifications: the transition of the single Sync has just $p_1 = p_{65}$ as its data constraint, whereas the corresponding transition in the product of 64 Syncs has $\exists p_{64}.\cdots\exists p_2.(p_1 = p_2 \wedge \cdots \wedge p_{64} = p_{65})$.

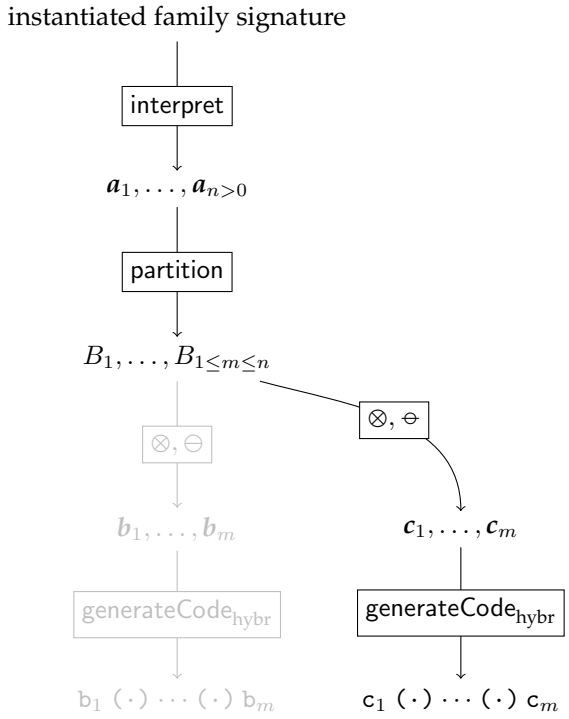


Figure 6.1: Hybrid compilation approach with syntactic subtraction

Clearly, solving the latter data constraint requires more resources than the former.

In the rest of this section, I develop a syntactic subtraction that, when applied 63 times to the product of 64 Syncs, yields the same data constraint as the one in the single Sync. Figure 6.1 shows the resulting compilation approach. First, I need a normalization for bringing constraint automata into a syntactically more convenient shape.

Normalization

Normalization consumes a constraint automaton $\boldsymbol{a}$ and produces a constraint automaton as output. Normalized constraint automata simplify proving certain properties. To normalize a constraint automaton, I manipulate only its data constraints, possibly splitting them into parts and distributing those parts over multiple transitions. Afterward, a normalized constraint automaton contains only *normal data constraints*. Every normal data constraint occurs in *prenex normal form* [Rau10a], in which zero or more existential quantifications envelop a quantifier-free *kernel*. The kernel of a normal data constraint consists of a conjunction of data literals.

Definition 38 (normal data constraints)**.** *A normal data constraint is an object generated by the following grammar:*

$$\begin{array}{lll} \ell & ::= & \textit{any data literal from Definition 15} \\ \varphi & ::= & \exists x.\varphi \mid \ell_1 \wedge \cdots \wedge \ell_{k\geq 1} \qquad \textit{(normal data constraints)} \end{array}$$

$\mathbb{DC}_{\exists,\wedge}$ *denotes the set of all normal data constraints.*

Clearly, $\mathbb{DC}$ in Definition 15 subsumes $\mathbb{DC}_{\exists,\wedge}$. Henceforth, let $\mathsf{Liter}(\varphi)$ denote the set of data literals in φ's kernel.

To normalize a constraint automaton $\boldsymbol{a}$, I must compute a new transition relation. For every transition (q, P, ϕ, q') in $\boldsymbol{a}$, I first compute a prenex normal form of ϕ, denoted by $\mathsf{pnf}(\phi)$. The goodness that ϕ exhibits by Definition 19 of $\mathbb{A}\text{UTOM}$ guarantees that such a prenex normal form always exists (i.e., the properties of goodness in Definition 17 guarantee that all inner existential quantifiers in ϕ can safely move outward). Next, I compute a *disjunctive normal form* of the kernel χ of $\mathsf{pnf}(\phi)$ [Rau10b], denoted by $\mathsf{dnf}(\chi)$. For every disjunct $\ell_1 \wedge \cdots \wedge \ell_k$ in $\mathsf{dnf}(\chi)$, I subsequently construct a new transition from state q to state q' labeled by synchronization constraint P and the normal data constraint built out of the existential quantifications in $\mathsf{pnf}(\phi)$ and $\ell_1 \wedge \cdots \wedge \ell_k$. Constructing new transitions in this way preserves the semantics of the original transition, because (i) prenex/disjunctive normal forms preserve the semantics of ϕ, (ii) existential quantification distributes over disjunction, and (iii) multiple transitions out of the same state represent a disjunction.

Definition 39 (normalization)**.** $|\cdot| : \mathbb{A}\text{UTOM} \to \mathbb{A}\text{UTOM}$ *denotes the function defined by the following equation:*

$$\begin{array}{c} |(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0))| = \\ (Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, |\longrightarrow|, (q^0, \mu^0)) \end{array}$$

where $|\longrightarrow|$ *denotes the smallest relation induced by the following rule:*

$$\frac{q \xrightarrow{P,\phi} q' \ \textbf{and} \ \mathsf{pnf}(\phi) = \exists x_1.\cdots\exists x_l.\chi \ \textbf{and} \ \mathsf{dnf}(\chi) = \phi_1 \vee \cdots \vee \phi_k}{q \ |\xrightarrow{P,\exists x_1.\cdots\exists x_l.\phi_{1\leq i\leq k}}| \ q'} \tag{6.1}$$

The following theorem states the correctness of Definition 39 of $|\cdot|$: normalization of a constraint automaton yields a behaviorally congruent constraint automaton. (Consequently, because $\simeq$ implies $\approx$ by Theorem 1, a normalized constraint automaton accepts the same language as its original.)

Theorem 12. $\boldsymbol{a} \simeq |\boldsymbol{a}|$

The following theorem states the effectiveness of Definition 39: a normalized constraint automaton indeed contains only normal data constraints.

Theorem 13. $\mathsf{Dc}(|\boldsymbol{a}|) \subseteq \mathbb{DC}_{\exists,\wedge}$

Syntactic Subtraction

I now proceed by defining syntactic substitution. I consider only (constraint automata with) normal data constraints, without loss of generality (because I can normalize every constraint automaton to a behaviorally congruent one).

First, I need to introduce the concept of *determinants* of free data variables in normal data constraints. For a normal data constraint φ and one of its free data variables $x \in \mathsf{Free}(\varphi)$, the set of determinants of x consists of those terms that *precisely* determine the datum $\sigma(x)$ assigned to x in any data assignment σ that satisfies φ (i.e., $\sigma \models \varphi$). "Precisely" here means that a determinant neither overspecifies nor underspecifies $\sigma(x)$. Thus, if a set of determinants contains multiple data terms, each of those data terms evaluates to the same datum under σ. Determinants furthermore determine $\sigma(x)$ independent of x itself: no determinant of x has x among its free data variables (i.e., determinants have no recursion). In the following definition, recall that a stands for a negated data atom in Definition 15.

Definition 40 (determinants)**.** $\mathsf{Determ} : \mathbb{X} \times \mathbb{DC}_{\exists,\wedge} \to 2^{\mathbb{TERM}}$ *denotes the function defined by the following equations:*

$$
\begin{aligned}
&\mathsf{Determ}_x(\top), \mathsf{Determ}_x(\bot) = \emptyset \\
&\mathsf{Determ}_x(\mathtt{K}(M)) = \emptyset \\
&\mathsf{Determ}_x(t_1 = t_2) = \begin{cases} \{t_2\} & \textbf{if } \left[t_1 = x \textbf{ and } x \notin \mathsf{Variabl}(t_2)\right] \\ \{t_1\} & \textbf{if } \left[t_2 = x \textbf{ and } x \notin \mathsf{Variabl}(t_1)\right] \\ \emptyset & \textbf{otherwise} \end{cases} \\
&\mathsf{Determ}_x(R(t_1, \ldots, t_k)) = \emptyset \\
&\mathsf{Determ}_x(\neg a) = \emptyset \\
&\mathsf{Determ}_x(\ell_1 \wedge \cdots \wedge \ell_k) = \mathsf{Determ}_x(\ell_1) \cup \cdots \cup \mathsf{Determ}_x(\ell_k) \\
&\mathsf{Determ}_x(\exists x'.\varphi') = \begin{cases} \mathsf{Determ}_x(\varphi) & \textbf{if } x \neq x' \\ \emptyset & \textbf{otherwise} \end{cases}
\end{aligned}
$$

For instance, let ϕ_{eg} denote the data constraint defined by the following equation (which occurs in the constraint automaton for the instantiated family signature `OddFibonacciPart(A,C;F,H)` in Figure 3.27):

$$\phi_{\text{eg}} = {}^{\bullet}\mathsf{x} = \mathsf{B} \wedge \mathsf{C} = \mathsf{D} \wedge \mathtt{add}(\mathsf{B}, \mathsf{D}) = \mathsf{E} \wedge \mathsf{E} = \mathsf{F} \wedge \mathsf{E} = \mathsf{G} \wedge \neg\mathtt{Odd}(\mathsf{G})$$

Because $\mathbb{DC}_{\exists,\wedge}$ contains ϕ_{eg}, I can directly apply Determ to compute the determinants of its free variables:

$$
\begin{aligned}
&\mathsf{Determ}_{{}^{\bullet}\mathsf{x}}(\phi_{\text{eg}}) = \{\mathsf{B}\} \\
&\mathsf{Determ}_{\mathsf{B}}(\phi_{\text{eg}}) = \{{}^{\bullet}\mathsf{x}\} \\
&\mathsf{Determ}_{\mathsf{C}}(\phi_{\text{eg}}) = \{\mathsf{D}\} \\
&\mathsf{Determ}_{\mathsf{D}}(\phi_{\text{eg}}) = \{\mathsf{C}\}
\end{aligned}
\qquad
\begin{aligned}
&\mathsf{Determ}_{\mathsf{E}}(\phi_{\text{eg}}) = \{\mathtt{add}(\mathsf{B}, \mathsf{D}), \mathsf{D}, \mathsf{D}\} \\
&\mathsf{Determ}_{\mathsf{F}}(\phi_{\text{eg}}) = \{\mathsf{E}\} \\
&\mathsf{Determ}_{\mathsf{G}}(\phi_{\text{eg}}) = \{\mathsf{E}\}
\end{aligned}
$$

Let $|\boldsymbol{a}|$ denote a normalized constraint automaton, and let φ denote one of its normal data constraints. Suppose that I subtract x from $|\boldsymbol{a}|$ with $\ominus$. By Definition 30 of $\ominus$, the transition(s) of $|\boldsymbol{a}|$ previously labeled by φ now carry $\exists x.\varphi$. However, if x has determinants, instead of enveloping φ in an existential quantification as $\ominus$ does, I can alternatively perform a syntactic substitution of one of those determinants for x. I formalize such a substitution with the following function.

Definition 41 (syntactic existential quantification)**.** $\mathsf{exists} : \mathbb{X} \times \mathbb{DC}_{\exists,\wedge} \to \mathbb{DC}_{\exists,\wedge}$ *denotes the function defined by the following equations:*

$$\mathsf{exists}_x(\varphi) = \begin{cases} \varphi[t/x] & \textbf{if } \left[\mathsf{Determ}_x(\varphi) \neq \emptyset \textbf{ and } t = \min(\mathsf{Determ}_x(\varphi))\right] \\ \exists x.\varphi & \textbf{otherwise} \end{cases}$$

In this definition, function $\min(\cdot)$ takes the least element in $\mathsf{Determ}_x(\phi)$, under the global order on data constraints $<_{\mathbb{DC}}$, to ensure that exists always produces the same output under the same input. The following equations exemplify the (nested) application of exists on ϕ_{eg} as defined above for all internal ports.

$$\begin{aligned}
&\mathsf{exists}_{\mathsf{G}}(\mathsf{exists}_{\mathsf{E}}(\mathsf{exists}_{\mathsf{D}}(\mathsf{exists}_{\mathsf{B}}(\phi_{\text{eg}})))) \\
&= \mathsf{exists}_{\mathsf{G}}(\mathsf{exists}_{\mathsf{E}}(\mathsf{exists}_{\mathsf{D}}(\mathsf{exists}_{\mathsf{B}}(\\
&\quad {}^{\bullet}\mathsf{x} = \mathsf{B} \wedge \mathsf{C} = \mathsf{D} \wedge \mathtt{add}(\mathsf{B},\mathsf{D}) = \mathsf{E} \wedge \mathsf{E} = \mathsf{F} \wedge \mathsf{E} = \mathsf{G} \wedge \neg\mathtt{Odd}(\mathsf{G})))))) \\
&= \mathsf{exists}_{\mathsf{G}}(\mathsf{exists}_{\mathsf{E}}(\mathsf{exists}_{\mathsf{D}}(\\
&\quad {}^{\bullet}\mathsf{x} = {}^{\bullet}\mathsf{x} \wedge \mathsf{C} = \mathsf{D} \wedge \mathtt{add}({}^{\bullet}\mathsf{x},\mathsf{D}) = \mathsf{E} \wedge \mathsf{E} = \mathsf{F} \wedge \mathsf{E} = \mathsf{G} \wedge \neg\mathtt{Odd}(\mathsf{G})))) \\
&= \mathsf{exists}_{\mathsf{G}}(\mathsf{exists}_{\mathsf{E}}(\\
&\quad {}^{\bullet}\mathsf{x} = {}^{\bullet}\mathsf{x} \wedge \mathsf{C} = \mathsf{C} \wedge \mathtt{add}({}^{\bullet}\mathsf{x},\mathsf{C}) = \mathsf{E} \wedge \mathsf{E} = \mathsf{F} \wedge \mathsf{E} = \mathsf{G} \wedge \neg\mathtt{Odd}(\mathsf{G}))) \\
&= \mathsf{exists}_{\mathsf{G}}(\\
&\quad {}^{\bullet}\mathsf{x} = {}^{\bullet}\mathsf{x} \wedge \mathsf{C} = \mathsf{C} \wedge \mathtt{add}({}^{\bullet}\mathsf{x},\mathsf{C}) = \mathsf{F} \wedge \mathsf{F} = \mathsf{F} \wedge \mathsf{F} = \mathsf{G} \wedge \neg\mathtt{Odd}(\mathsf{G})) \\
&= \quad {}^{\bullet}\mathsf{x} = {}^{\bullet}\mathsf{x} \wedge \mathsf{C} = \mathsf{C} \wedge \mathtt{add}({}^{\bullet}\mathsf{x},\mathsf{C}) = \mathsf{F} \wedge \mathsf{F} = \mathsf{F} \wedge \mathsf{F} = \mathsf{F} \wedge \neg\mathtt{Odd}(\mathsf{F})
\end{aligned}$$

I define syntactic subtraction in terms of exists.

Definition 42 (syntactic subtraction)**.** $\ominus : \mathbb{A}\text{UTOM} \times \mathbb{P} \to \mathbb{A}\text{UTOM}$ *denotes the function defined by the following equation:*

$$\begin{aligned}
&(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0)) \ominus p = \\
&\quad (Q, (P^{\text{all}} \setminus \{p\}, P^{\text{in}} \setminus \{p\}, P^{\text{out}} \setminus \{p\}), M, \longrightarrow_{\ominus}, (q^0, \mu^0))
\end{aligned}$$

where $\longrightarrow_{\ominus}$ *denotes the smallest relation induced by the following rules:*

$$\frac{q \xrightarrow{P,\phi} q' \textbf{ and } \phi \in \mathbb{DC}_{\exists,\wedge}}{q \xrightarrow{P\setminus\{p\},\mathsf{exists}_p(\phi)}_{\ominus} q'} \quad (6.2) \qquad \frac{q \xrightarrow{P,\phi} q' \textbf{ and } \phi \notin \mathbb{DC}_{\exists,\wedge}}{q \xrightarrow{P\setminus\{p\},\exists p.\phi}_{\ominus} q'} \quad (6.3)$$

In the previous definition, I use exists to remove ports from data constraints. Although Definition 41 of exists also allows for removing data variables for memory cells, I do not pursue such syntactic subtraction in this thesis.

Before I can actually adopt the compilation approach in Figure 6.1 in practice, I must establish the correctness and effectiveness of syntactic subtraction. I consider syntactic subtraction correct if it yields a behaviorally congruent—hence, behaviorally equivalent by Theorem 1—constraint automaton to the constraint automaton that semantic subtraction would have yielded. Before formulating this property as a theorem, the following lemma first states the equivalence of existential quantification and exists.

Lemma 16. $\exists x.\varphi \equiv \mathsf{exists}_x(\varphi)$

From Lemma 16, I conclude the following correctness theorem.

Theorem 14. $\boldsymbol{a} \ominus p \simeq \boldsymbol{a} \oplus p$

Note that this theorem works not only for normalized constraint automata but also for arbitrary ones. Normalization plays a role only in the sequel, where I show that syntactic subtraction has its intended effect when applied to normalized constraint automata.

I consider syntactic subtraction effective if, after syntactically subtracting a port p from a normalized constraint automaton $|\boldsymbol{a}|$, that port no longer occurs in any of that automaton's data constraints. Generally, however, such unconditional effectiveness does not hold true. After all, if $|\boldsymbol{a}|$ has a data constraint φ in which p occurs, but p has no determinants in φ, syntactic subtraction has nothing to replace p with. In that case, $\mathsf{exists}_p(\varphi) = \exists p.(\varphi)$, and consequently, syntactic subtraction does not have its intended effect. Fortunately, syntactic subtraction does satisfy a weaker—but still useful—form of effectiveness. To formulate this as a theorem, I first define a function that computes *ever-determined ports*. Under a set of excluded data terms T, I consider a port p ever-determined in a constraint automaton $\boldsymbol{a}$ iff both p occurs in $\boldsymbol{a}$ and every data constraint in $\boldsymbol{a}$ has a determinant for p outside T.

Definition 43 (ever-determined ports)**.** $\mathsf{Edp} : 2^{\mathbb{TERM}} \times \mathbb{AUTOM} \to 2^{\mathbb{P}}$ *denotes the function defined by the following equation:*

$$\mathsf{Edp}_T(\boldsymbol{a}) = \{p \mid [\begin{bmatrix} p \in \mathsf{Variabl}(\phi) \\ \textbf{and} \;\; \phi \in \mathsf{Dc}(\boldsymbol{a}) \end{bmatrix} \;\; \textbf{implies} \;\; \mathsf{Determ}_p(\phi) \setminus T \neq \emptyset] \;\; \textbf{for all} \;\; \phi\}$$

For instance, p_1, p_2, and p_3 all qualify as ever-determined (under $\emptyset$) in members of Merger2 in Figure 3.4. To understand the ever-determinedness of p_1, observe that p_1 occurs in the data constraint on the top transition in Merger2 and that p_1 has a determinant outside $\emptyset$ in that data constraint (namely p_3); because p_1 does not occur in the data constraint on the bottom transition in Merger2, p_1 indeed qualifies as ever-determined. A similar explanation applies to p_2. To understand the ever-determinedness of p_3, observe that p_3 occurs in the data constraint on both transitions in Merger2 and that p_3 has a determinant outside $\emptyset$ in both these data constraints (namely p_1 and p_2). Consequently, also p_3 qualifies as ever-determined. In contrast, p_2 in members of Filter in Figure 3.4

does *not* qualify as ever-determined (under any set of excluded data terms), because p_2 occurs in the data constraint on the top transition in Filter but does not have a single determinant in that data constraint.

The following theorem states the effectiveness of syntactic subtraction, conditional on ever-determinedness: after syntactically subtracting an ever-determined port from a normalized constraint automaton, that port no long occurs in any of that automaton's data constraints.

Theorem 15.

$$p \in \mathsf{Edp}_T(|\boldsymbol{a}|) \textbf{ implies } p \notin \{x \mid \phi \in \mathsf{Dc}(|\boldsymbol{a}| \ominus p) \textbf{ and } x \in \mathsf{Variabl}(\phi)\}$$

A FOCAML compiler can check ports for ever-determinedness before applying syntactic subtraction. To reduce the number of such checks, however, I also present a conjecture about preservation of ever-determinedness by operations on constraint automata. First, note that for syntactic subtraction of port p' in constraint automaton $\boldsymbol{a}$ to preserve the ever-determinedness of a port p, every data constraint in $\boldsymbol{a}$ should have a non-p' determinant for p. For instance, in E = F, port E has port F as its determinant but not so in $\mathsf{exists}_{\mathsf{F}}(\mathsf{E} = \mathsf{F}) = \mathsf{E} = \mathsf{E}$. Thus, syntactic subtraction of F in a constraint automaton with a transition labeled by E = F does not preserve the ever-determinedness of E. In contrast, syntactic subtraction of F in a constraint automaton with a transition labeled by $\mathsf{E} = \mathsf{F} \wedge \mathsf{E} = \mathsf{G}$ *does* preserve the ever-determinedness of E, because E has a non-F determinant in that data constraint, namely G. The following conjecture makes this precise.

Conjecture 1.

- $p \in \mathsf{Edp}_T(|\boldsymbol{a}_1|) \textbf{ implies } p \in \mathsf{Edp}_T(||\boldsymbol{a}_1| \otimes |\boldsymbol{a}_2||)$
- $p \in \mathsf{Edp}_T(|\boldsymbol{a}|) \textbf{ implies } p \in \mathsf{Edp}_T(|\boldsymbol{a}| \ominus p')$
- $p \in \mathsf{Edp}_{T \cup \{p'\}}(|\boldsymbol{a}|) \textbf{ implies } p \in \mathsf{Edp}_T(|\boldsymbol{a}| \ominus p')$

The first item of this conjecture states that multiplication preserves ever-determinedness; its second and its third item state that also subtraction preserves ever-determinedness under a suitable set T.

For now, I leave these preservation properties as a conjecture, because its truth or falsehood does not matter much in practice: although its (dis)proof would yield more insight in the theory of syntactic subtraction, practical consequences remain insignificant. After all, this conjecture helps only in *predicting* when syntactic subtraction may have its intended effect; it does not affect syntactic subtraction's correctness whatsoever. Still, if this conjecture indeed holds true as I strongly suspect, a FOCAML compiler can reduce its number of checks for ever-determinedness as explained next.

All instances of the primitives in Figure 3.4 have only ever-determined output ports and only ever-determined internal ports (vacuously, because none of these constraint automata has any internal ports). Regardless of whether these

primitives have ever-determined input ports—some of them do, others do not—the ever-determinedness of their output and internal ports already suffices for syntactic subtraction to have its intended effect whenever a FOCAML compiler subtracts only internal ports (as it always does). To see this, recall from Definition 29 of $\otimes$ that new internal ports arise only through multiplication. In particular, the set of internal ports in a product contains (i) the internal ports in its multiplicands and (ii) those multiplicands' shared ports, where every shared port *must* serve as an output port in exactly one multiplicand. Now, if two multiplicands indeed have only ever-determined output ports and only ever-determined internal ports (as all instances of the primitives in Figure 3.4 do), this means two things. First, by Conjecture 1, the product has only ever-determined internal ports, namely the multiplicands' shared output ports and their internal ports. Consequently, syntactic subtraction has its intended effect on all these ports. Second, as the multiplicands, also the product has only ever-determined output ports, namely the multiplicands' unshared output ports. Consequently, this reasoning can recur. Thus, if a FOCAML compiler has established that all primitives in a core set—not necessarily the same core set as in Figure 3.4—have only ever-determined output ports and only ever-determined internal ports, it does not need to check those ports for ever-determinedness again in any of those primitives' products. Symmetrically, this also works for input ports instead of output ports.

Incidentally, if I apply syntactic subtraction to the sequence of 64 Syncs as in the beginning of this chapter, and after removing $x = x$ literals (each of which trivially equates to $\top$), I get *exactly* the same data constraint as the one in the single Sync. Using syntactic subtraction, thus, the sequence of 64 Sync has the same performance as the single Sync, as shown in more detail in Section 6.2.

6.2 Practice

(I have not yet submitted the material in this section for publication.)

Compiler

I extended Lykos with the ability to apply syntactic subtraction as in Figure 6.1, controllable through flag `SUBTRACT_SYNTACTICALLY`. When raised, Lykos first checks whether all primitives in the core set have either only ever-determined output ports or, symmetrically, only ever-determined input ports. If so, by Conjecture 1, Lykos does not need to check ports in products for ever-determinedness during the compilation process. Note that even if Conjecture 1 turns out not to hold true, Lykos does not generate faulty code: in the worst case, Lykos unsuccessfully tries to syntactically subtract p without first checking p for ever-determinedness, but once such subtraction fails (in which case it cannot find a determinant for p), Lykos simply defaults to semantic subtraction, as in Definition 41. If this happens, compilation just takes a little longer. In contrast to Lykos' internals, the run-time library requires no modifications. Also,

code generated with syntactic subtraction looks very similar to code generated without syntactic subtraction, differing only in the size of data constraints (but the constraint solver in the run-time library solves them in exactly the same way).

Experiments I: Protocols

I repeated the same experiments as in Chapter 5 (and Chapter 4), generating code for members of families SyncK, FifoK, Merger, Router, LateAsyncMerger, EarlyAsyncMerger, OddFibonacci, and Chess with the `SUBTRACT_SYNTACTICALLY`-flag raised, but otherwise under the same conditions as in Chapter 5 (except for OddFibonacci, whose members I compiled under the Centralized Approach to manually avoid overparallelization). Figure 6.2 shows the per-family experimental results, averaged over five runs. The solid lines represent the actual measurements; the dotted lines represent inverse-proportional growth with respect to $k = 1$. The yellow lines represent the new results; the blue and red lines represent the results from Chapters 4 and 5. I compare the new results for OddFibonacci with its results in Chapter 4 instead of those in Chapter 5 because of the overparallelization issue discussed at the end of Chapter 5.

Figure 6.3 shows per-family speedup charts corresponding to the measurements in Figure 6.2; the dotted lines represent equal performance. For all constraint automata with which I experimented, syntactic subtraction indeed improves performance, to a lesser or to a greater extent. For members of SyncK, with syntactic subtraction, performance becomes constant in k (i.e., the number of Syncs that constitute a SyncK), exactly as one would expect of a neutral element for multiplication. Speedups grow linearly in k, up to 200% for $k = 64$. For members of FifoK, speedup stays roughly constant in k, at about 3%. Performance itself, in contrast, does not stay constant in k but degrades more or less linearly (from $k = 3$ onward). As for members of SyncK, inverse-proportional growth does not imply good scalability for members of FifoK: in principle, the size of a buffer (controlled by k) should not affect the speed with which data can pass through this buffer. Thus, for compiler-generated code for members of Fifo to scale well, its performance should stay constant in k instead of degrading linearly. In perhaps the simplest approach to achieve such constant performance, a compiler recognizes sequences of k consecutive Fifos in multiplication expressions and subsequently gives such sequences a special treatment, effectively generating special optimized code for k-capacity buffers. Although not a general method, sequences of Fifos occur frequently enough to justify an optimization as this. Nevertheless, I do not pursue this optimization in this thesis—it may constitute an interesting BSc project though.

For members of of LateAsyncMerger and EarlyAsyncMerger (to greater extent) and of Merger (to lesser extent), performance seems to approach constant in k (i.e., the number of producers). Members of those families do not require the producers to synchronize with each other nor with the consumer. Consequently, in principle, the performance of compiler-generated code for members of those families should not depend on the number of producers: a per-

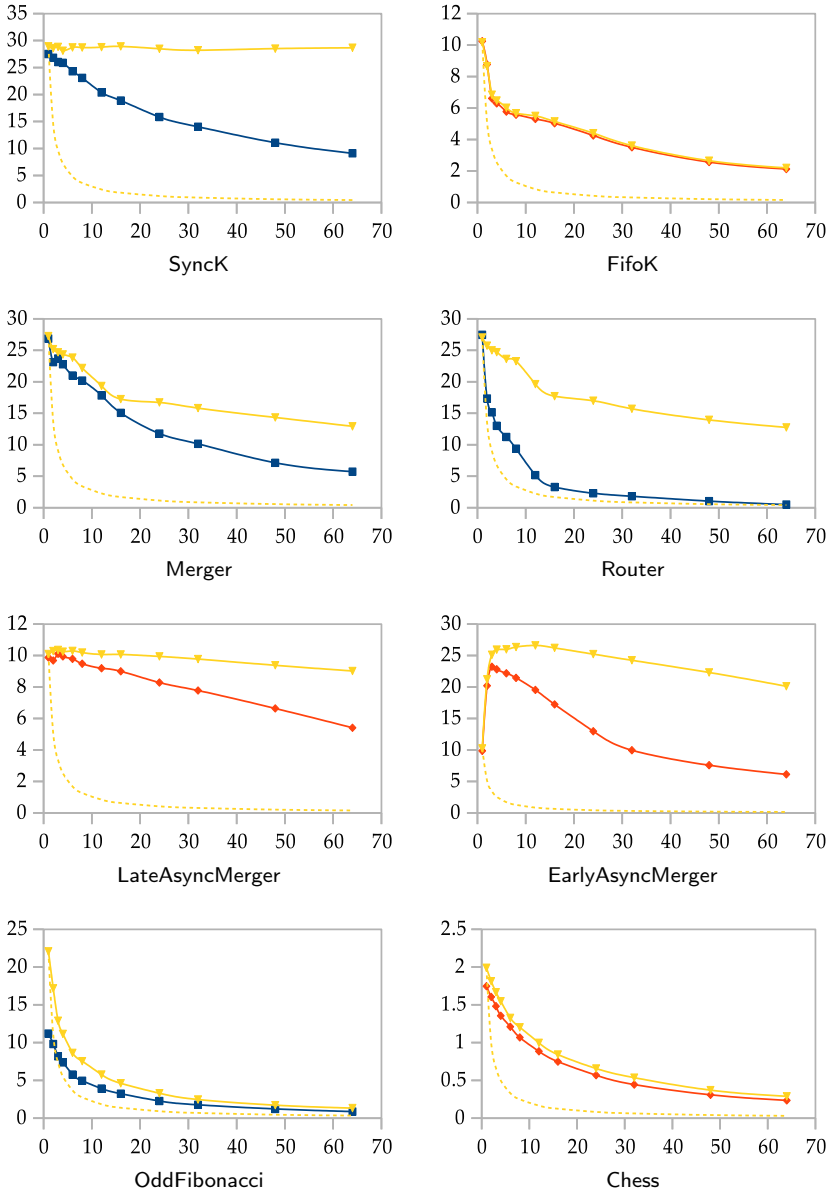


Figure 6.2: Performance (in number of completed rounds per four minutes) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

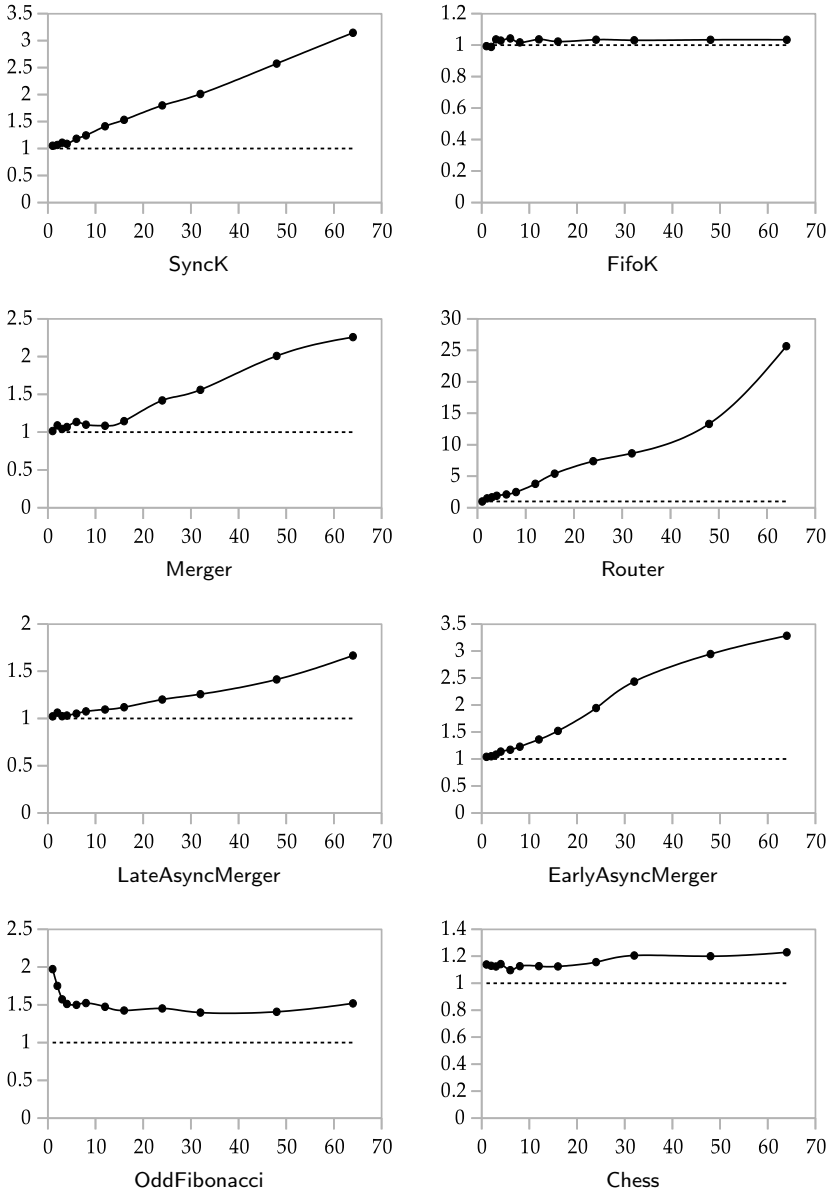


Figure 6.3: Speedup (relative to compiler-generated code in Chapter 5) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

fect compiler generates code with performance constant in k. As Sync and Fifo, thus, Merger, LateAsyncMerger, and EarlyAsyncMerger exemplify that inverse-proportionality forms a necessary condition for good scalability but not necessarily a sufficient one. In practice, however, the number of producers *does* influence the performance of code generated for members of Merger, LateAsyncMerger, and EarlyAsyncMerger, because those producers contend for the same "resource"—the consumer. Therefore, near-constant growth in k seems fair enough. My extension of Lykos with syntactic subtraction makes a significant step in achieving that goal, providing speedup of up to 125% for members of Merger, 67% for members of LateAsyncMerger and up to 228% for members of EarlyAsyncMerger. The improvements that I introduce in Chapters 7 and 8 push scalability even further toward a flat line.

In principle, a similar analysis as for members of Merger, LateAsyncMerger, and EarlyAsyncMerger applies to members of Router. However, Routers achieve truly spectacular speedup, of up to 2464%. This shows that without syntactic subtraction, Routers have complex data constraints, with many free data variables, whose constraint solving requires a very substantial amount of computational resources. Moreover, with syntactic subtraction, code generated for Mergers and Routers has similar performance (i.e., compare the yellow lines for Merger and Router in Figure 6.2), whereas without syntactic subtraction, code generated for Mergers outperforms code generated for Routers (i.e., compare the blue lines for Merger and Router in Figure 6.2). As Routers form just the inverse—in terms of port directions—of Mergers as explained in Chapter 3, it makes sense for code generated for Routers to have similar performance as code generated for Mergers. Their previously measured differences in performance therefore actually indicated a significant deficiency of Lykos (similar to its previous inability to properly handle the neutrality of Syncs). Syntactic subtraction solves this problem, providing similar performance for code generated for Mergers and Routers, as one may expect.

In contrast to members of Merger, Router, LateAsyncMerger, and EarlyAsyncMerger, members of OddFibonacci and Chess require workers to globally synchronize with everybody (instead of pairwise synchronous or asynchronous interaction). For those families, thus, one may reasonably expect the number of workers to affect performance, as long as performance does not degrade below inverse-proportionality. Although not clearly visible in Figure 6.2 (because of the scale on the y-axis), Figure 6.3 shows that syntactic subtraction leads to significant speedup also for members of OddFibonacci and Chess.

Experiments II: Programs

I repeated the same experiments as in Chapter 5 (and Chapter 4), generating code for the NPB benchmarks with the `SUBTRACT_SYNTACTICALLY`-flag raised, but otherwise under the same conditions as in Chapter 5. Using syntactic subtraction, in contrast to semantic subtraction as used in Chapter 5, Lykos succeeded in generating code for all values of k for all NPB benchmarks.

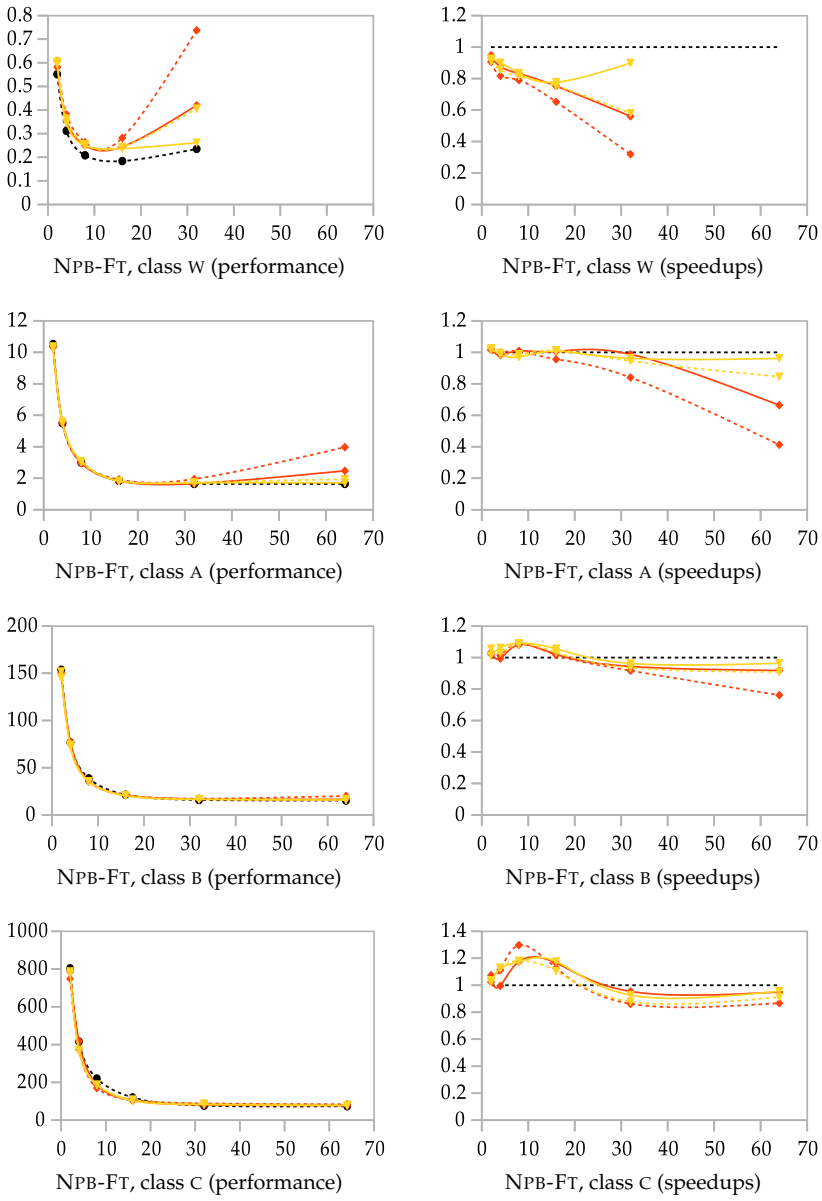


Figure 6.4: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

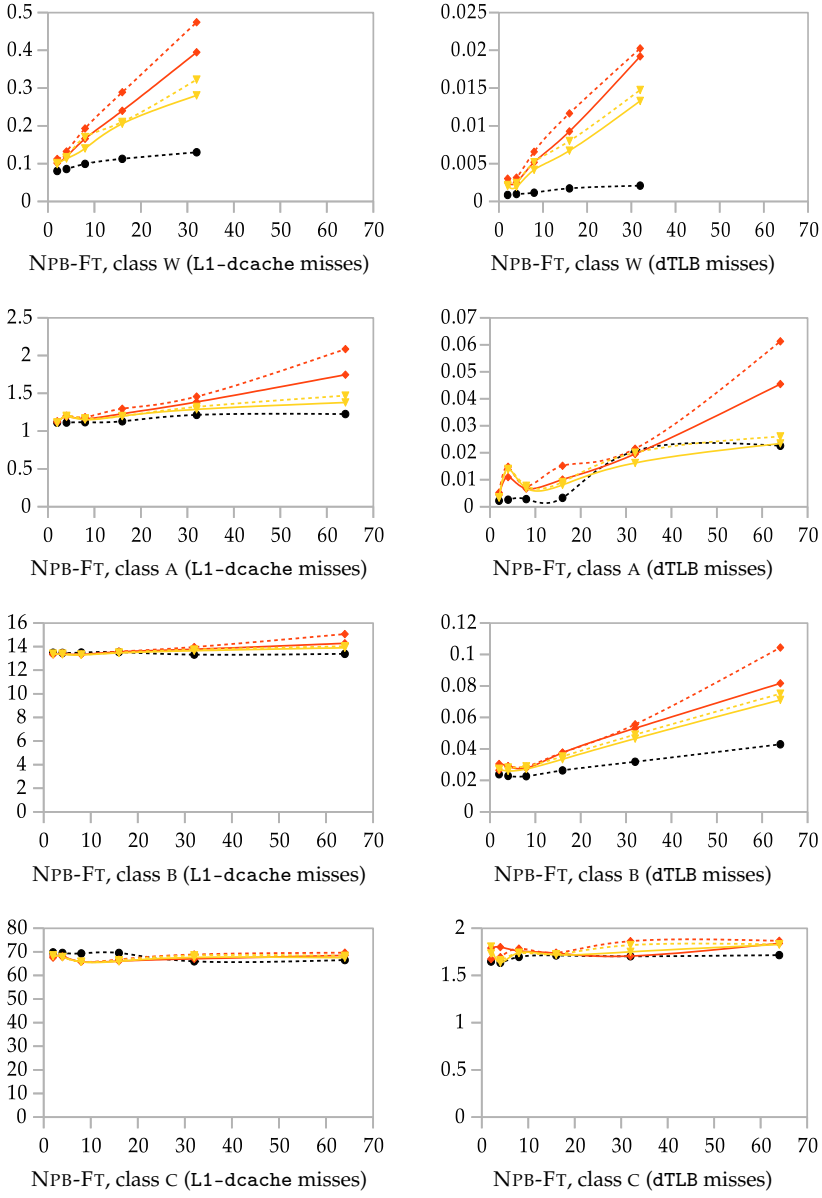


Figure 6.5: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

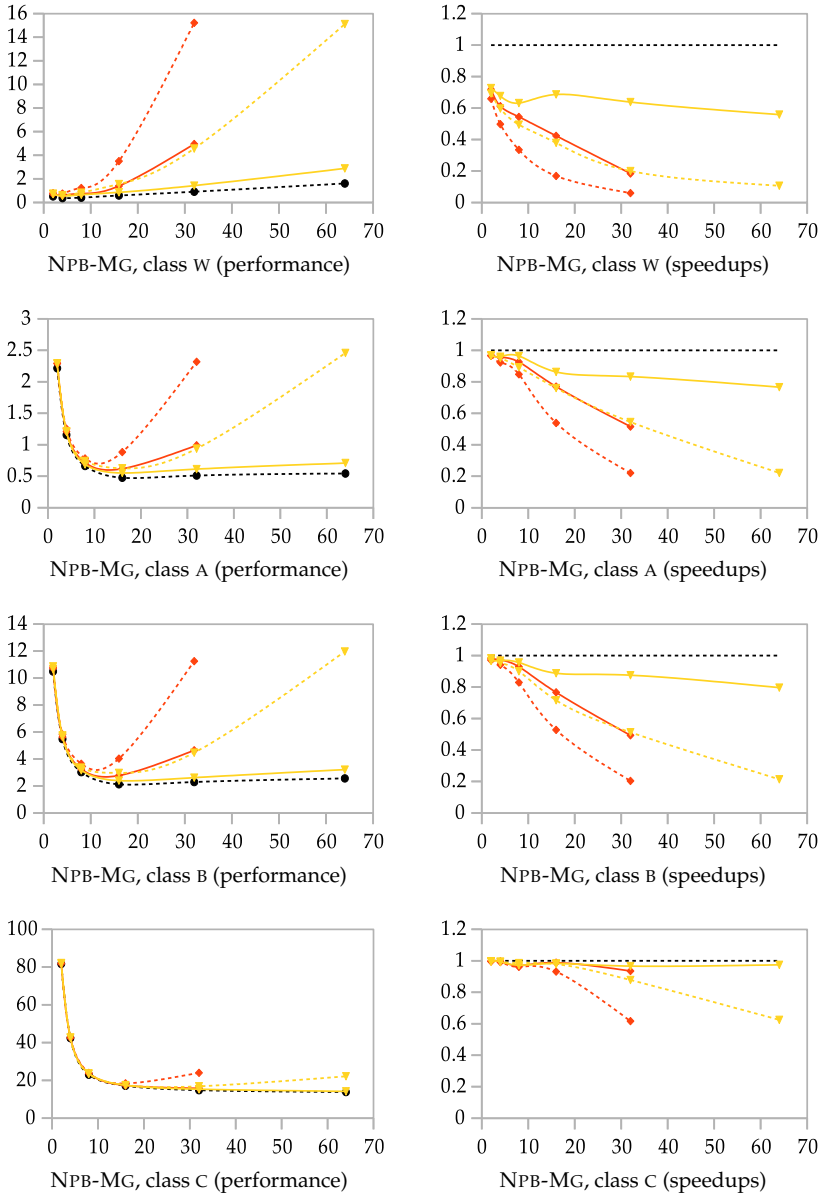


Figure 6.6: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

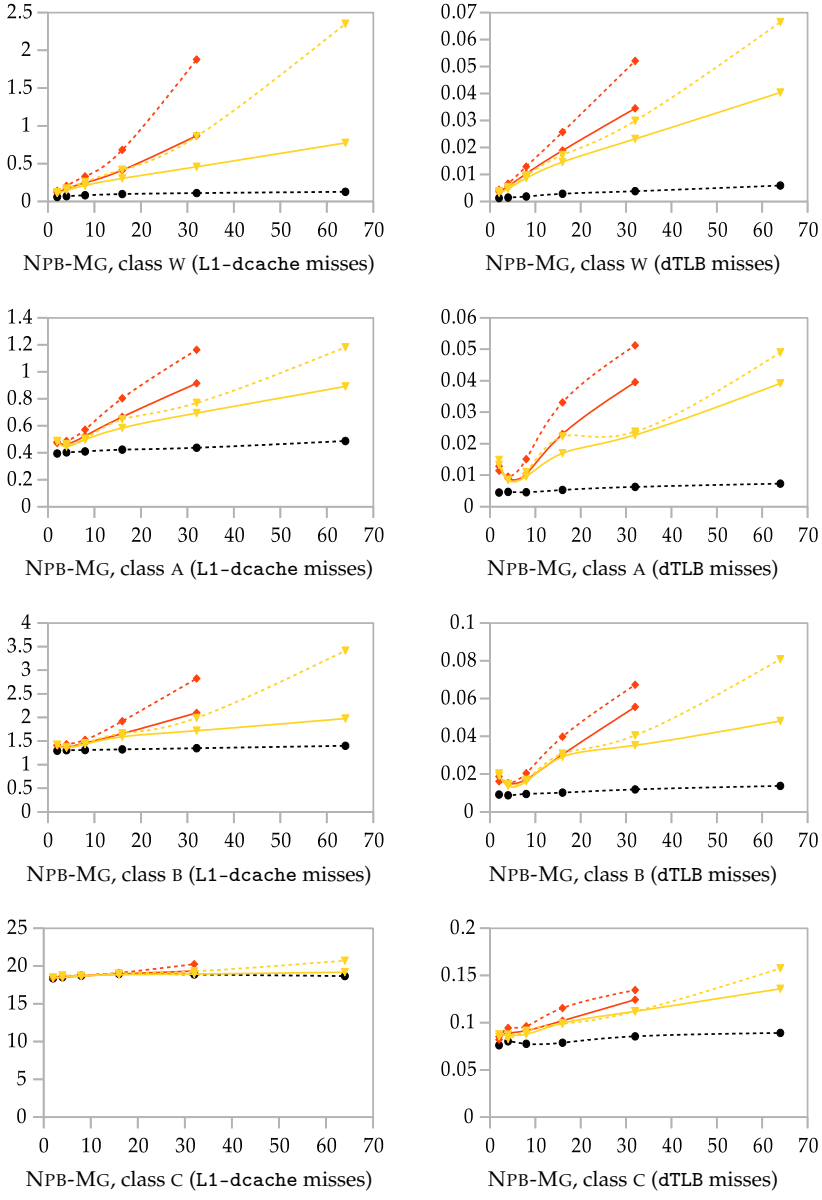


Figure 6.7: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

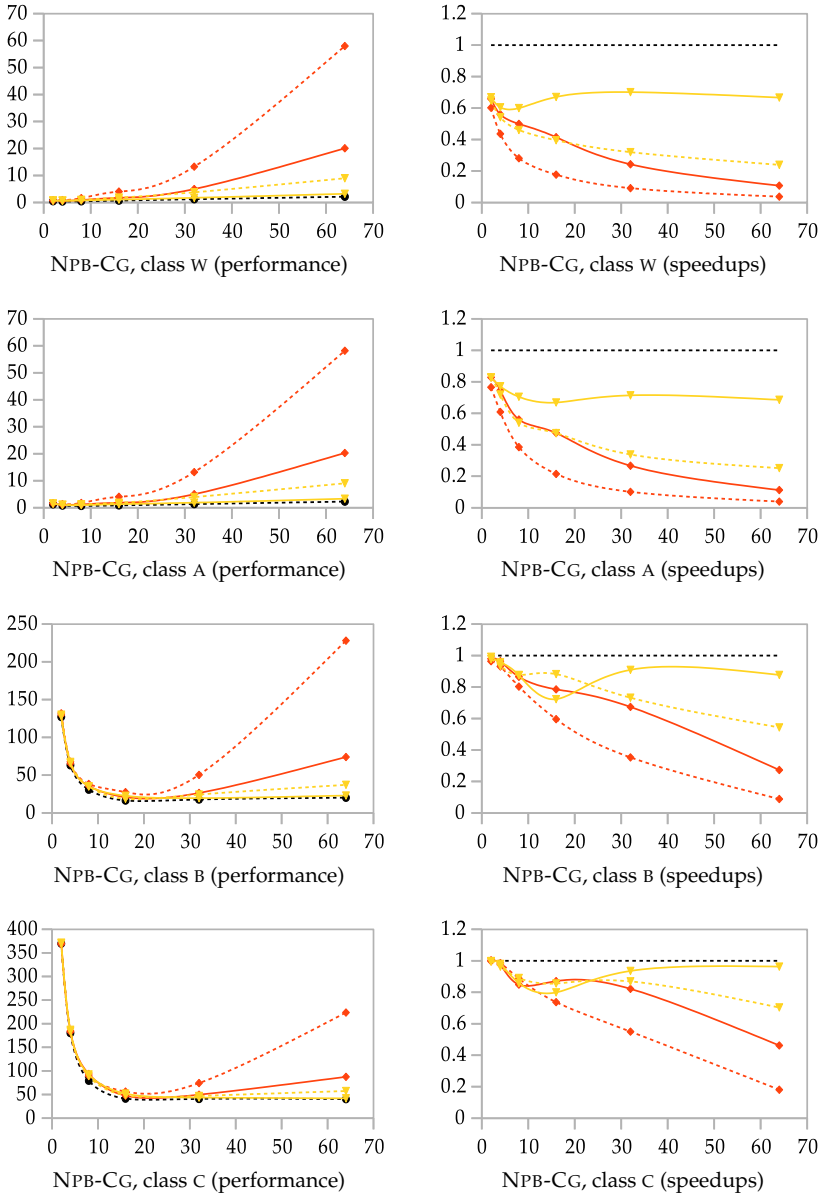


Figure 6.8: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

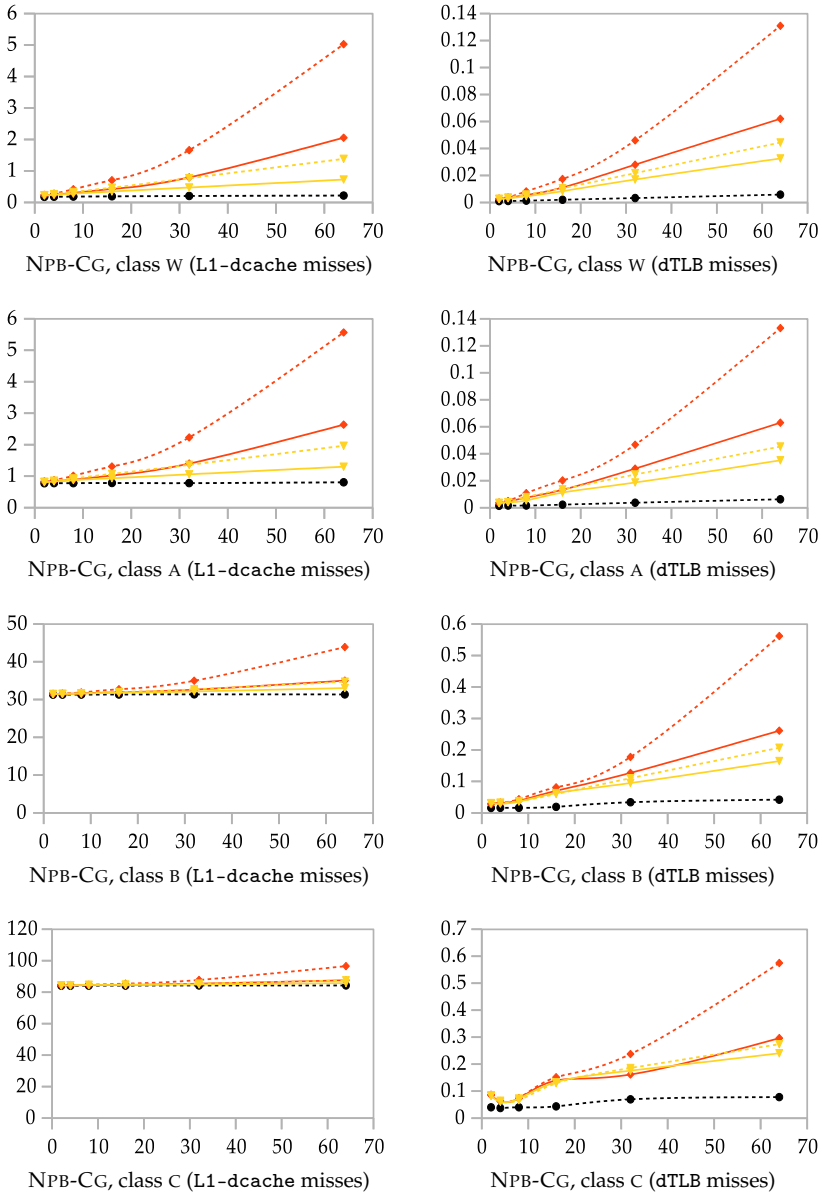


Figure 6.9: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

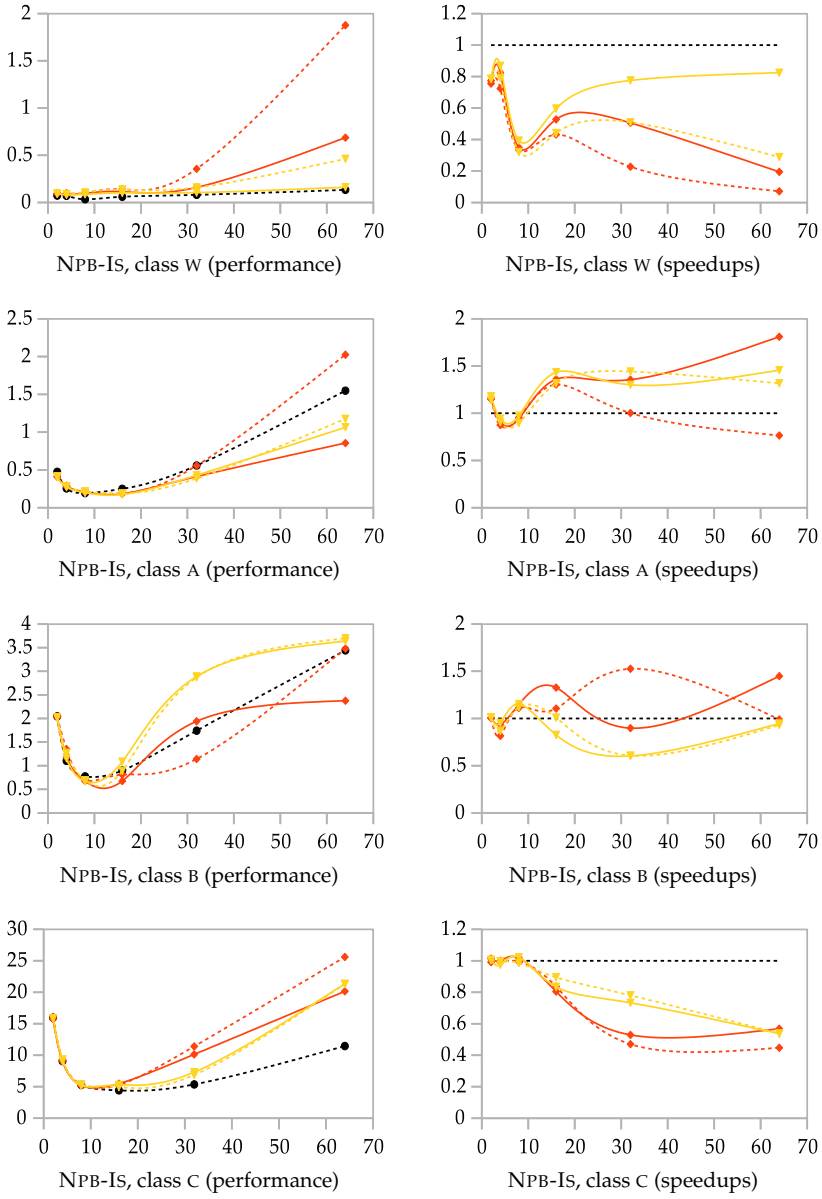


Figure 6.10: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

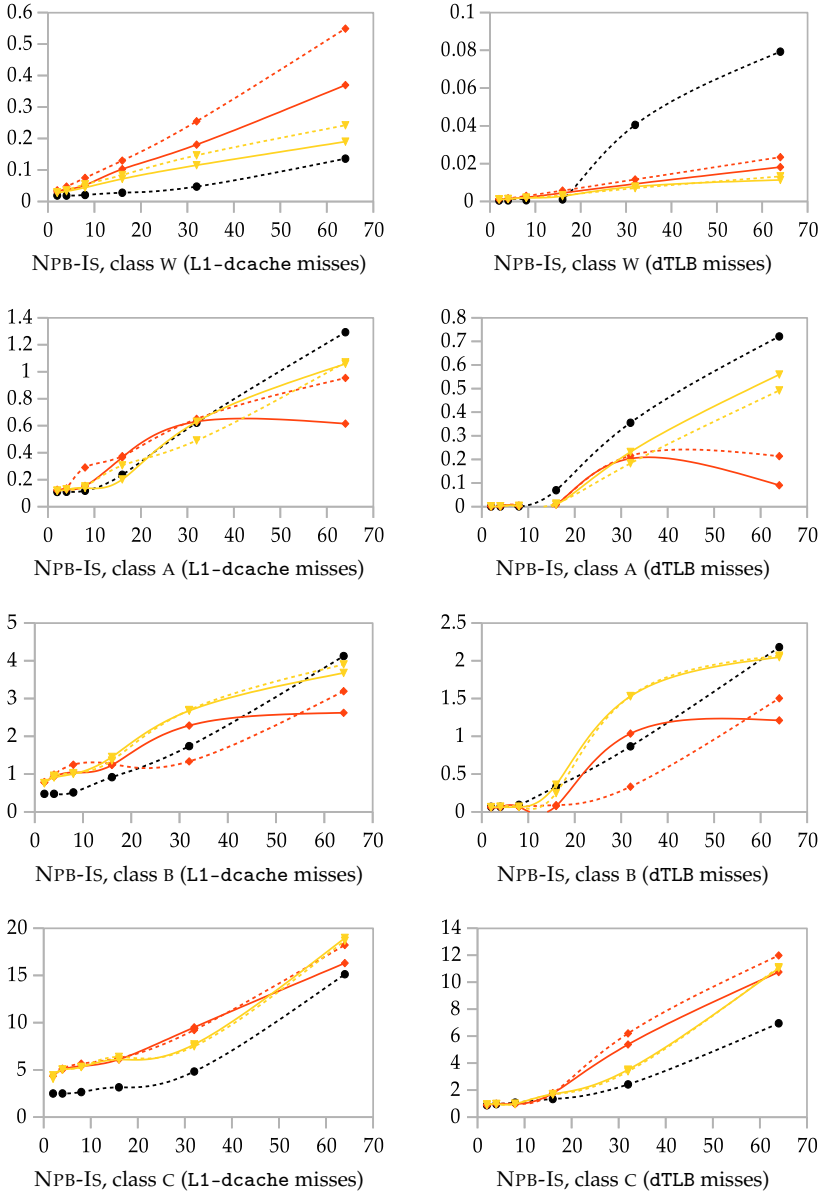


Figure 6.11: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

Figures 6.4–6.11 show performance charts for the FOCAML-to-Java-compiled versions of the NPB kernel benchmarks (averaged over five runs), speedup charts (with respect to their Java versions by Frumkin et al.), and charts about cache misses. The dotted yellow/red lines represent the MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks with and without syntactic subtraction (i.e., the yellow lines represent the new results, while the red lines represent the results in Chapter 5); the solid yellow/red lines represent the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions; the dotted black lines represent the Java versions by Frumkin et al.

I make the following main observations about these experimental results:

- Overall, the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks outperform their MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions (solid lines versus dotted lines), as in Chapter 5. Furthermore, overall, the FOCAML-to-Java-compiled versions with syntactic subtraction perform at least as well as the FOCAML-to-Java-compiled versions without syntactic subtraction (yellow lines versus red lines), and often better.

- Although the Java versions of the NPB kernel benchmarks by Frumkin et al. still slightly outperform many of their FOCAML-to-Java-compiled versions, the margin has decreased substantially (compared to the results in Chapter 5): syntactic subtraction really makes an impact in these NPB kernel benchmarks.

- The same point about cache misses made in Chapter 5 applies here too: numbers of cache misses seem a fair indicator of performance.

- The same point about increasing problem sizes made in Chapter 5 applies here too: as the problem size increases, the speedup generally improves.

- As in Chapter 5, differences in numbers of cache misses explain why the FOCAML-to-Java-compiled versions of NPB-IS without syntactic subtraction outperform their supposedly improved FOCAML-to-Java-compiled versions with syntactic subtraction in class A, $k = 64$, class B, $k \in \{32, 64\}$, and class C, $k = 64$.

Figures 6.12–6.17 show performance charts for the FOCAML-to-Java-compiled versions of the NPB application benchmarks (averaged over five runs), speedup charts (with respect to their Java versions by Frumkin et al.), and charts about cache misses. The lines have the same meaning as in the figures with experimental results for the NPB kernel benchmarks. Recall from Figure 5.12 that NPB-BT and NPB-LU do not support more than 22 and 31 slaves in class W, for which reason I have no measurements beyond $k = 16$ in class W for those benchmarks. In the same figure, note that NPB-BT, NPB-SP, and NPB-LU support at most 62 workers in class A. For that reason, I compiled

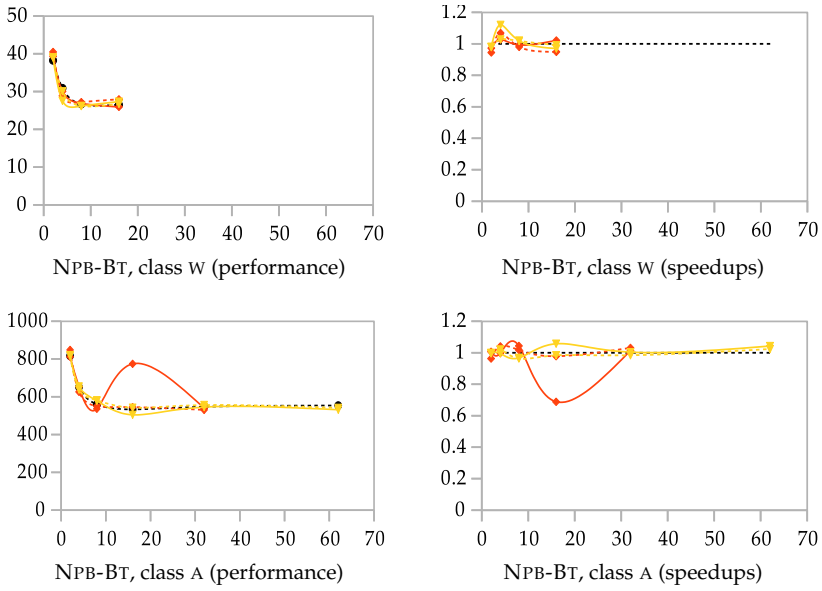


Figure 6.12: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

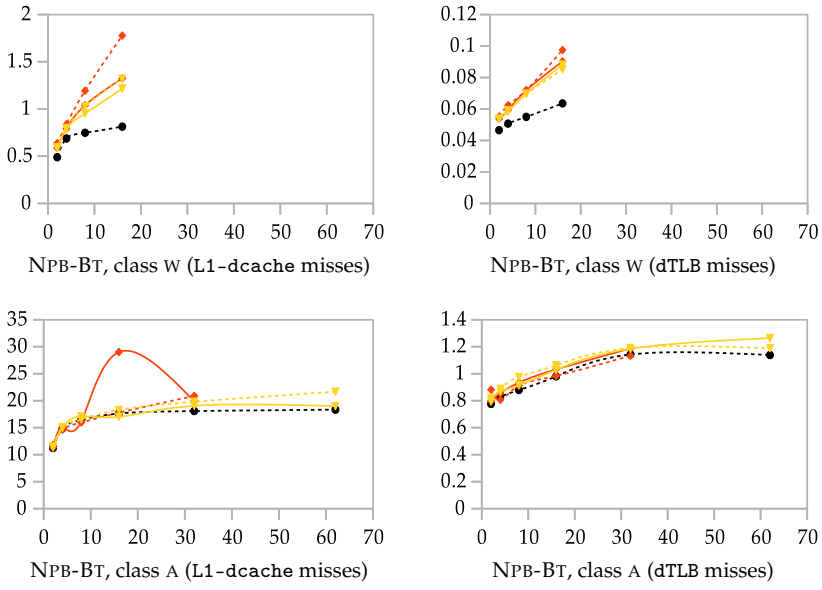


Figure 6.13: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

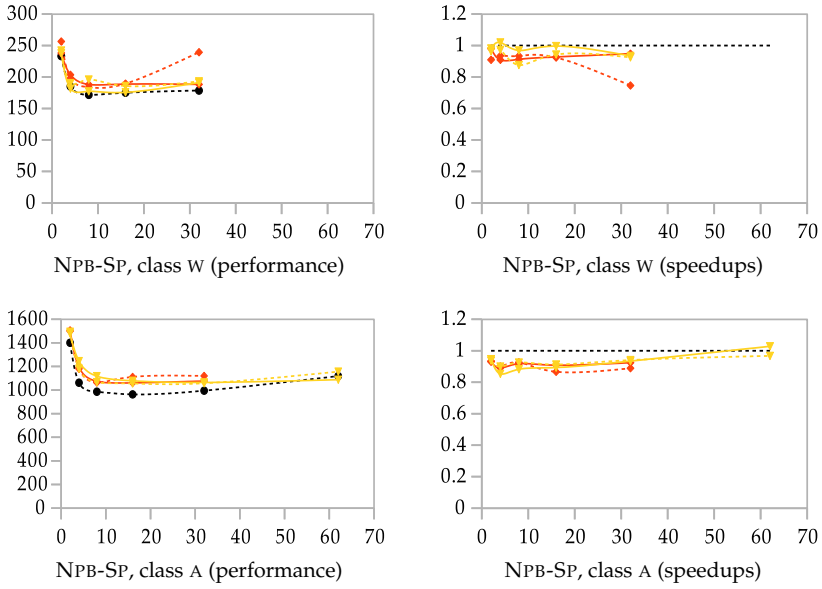


Figure 6.14: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

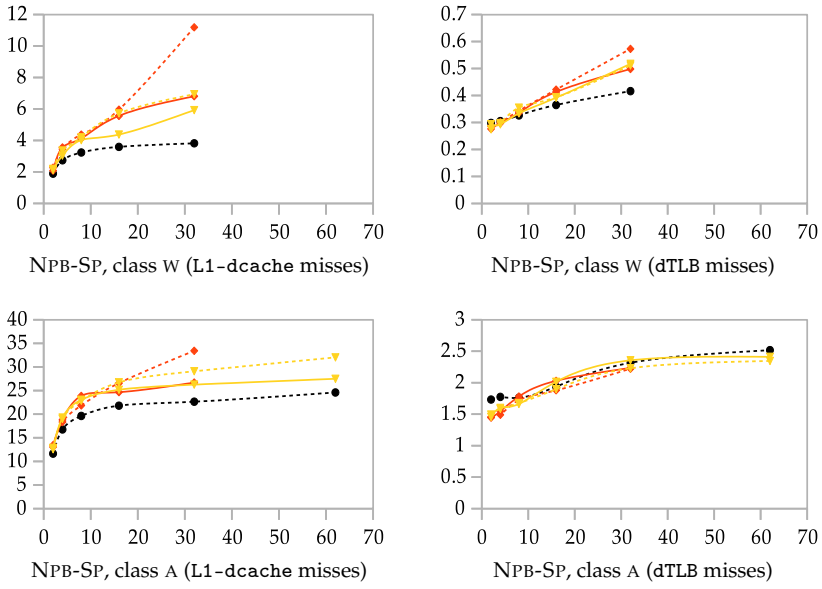


Figure 6.15: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

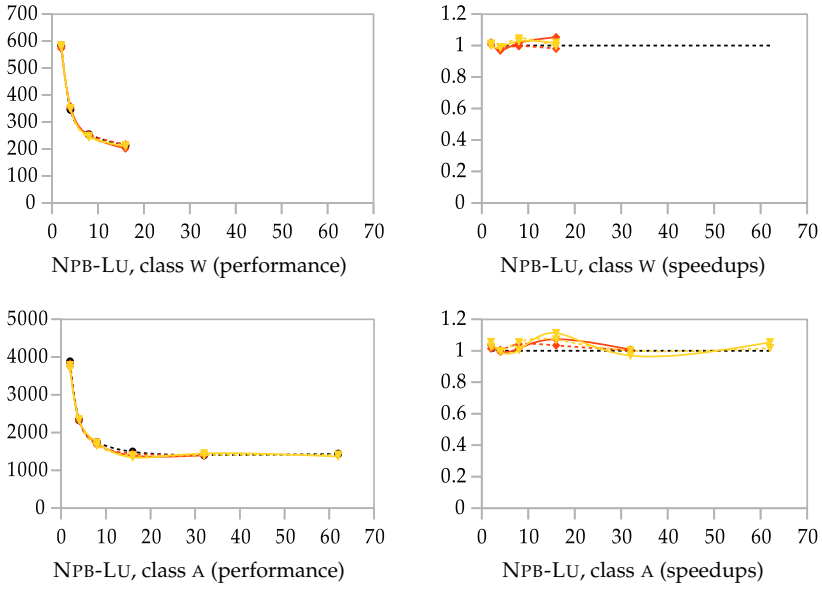


Figure 6.16: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

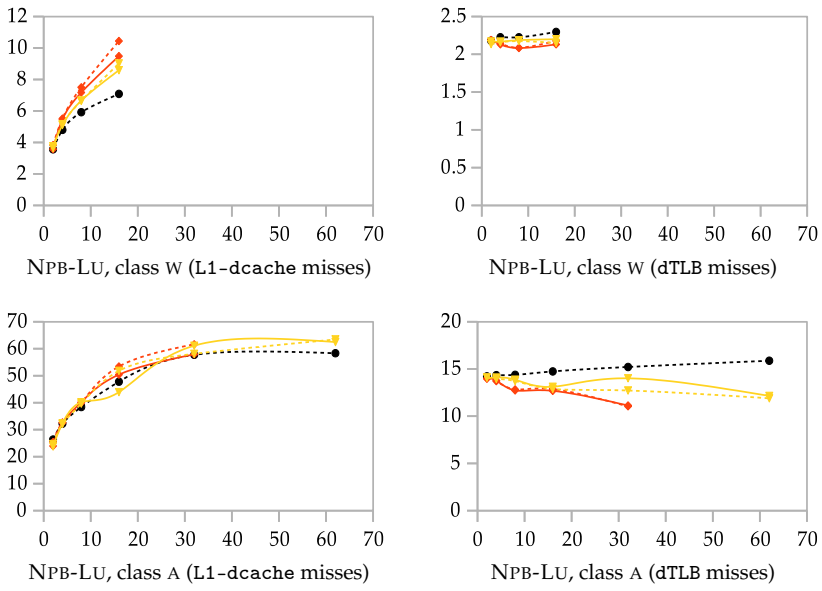


Figure 6.17: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

the FOCAML versions of those benchmarks for $k = 62$ instead of $k = 64$. Essentially, the same observations apply here as for the previous experimental results of the NPB kernel benchmarks.

Chapter 7

Improved Compilation III: Commandification

The experimental results in Chapter 6 show that syntactic subtraction already significantly improves performance. In this chapter, I present a complementary technique to further improve the data constraint checks involved in determining whether a transition can fire and, in particular, the expensive constraint solver calls made during such checks. Essentially, this new technique comprises the generation of a little, dedicated constraint solver for every data constraint at compile-time. At run-time, then, instead of calling a general-purpose constraint solver to check a data constraint, a protocol unit calls the more efficient constraint solver generated specifically for that data constraint.

In Section 7.1, I first introduce a basic sequential language in which to formalize these dedicated constraint solvers, called *data commands*. Subsequently, I present a translation of data constraints into data commands. Finally, I formally show that I can replace data constraints with their corresponding data commands in transition labels of constraint automata. In Section 7.2, I present an improved version of Lykos using the translation of data constraints into data commands, including new experimental results on performance.

Although the improvement presented in this chapter eventually results in improved compiler-generated code, as in Chapters 5 and 6, I define this improvement at the higher level of constraint automata instead of at the lower level of GPL code. Not only does this facilitate more elegant formal reasoning about correctness (compared to reasoning directly about GPL code), but it also eases the automatic application of this improvement by a FOCAML compiler. Moreover, it makes this improvement independent of GPLs—Java in this thesis—so that the same optimization automatically applies to, for instance, generated C code.

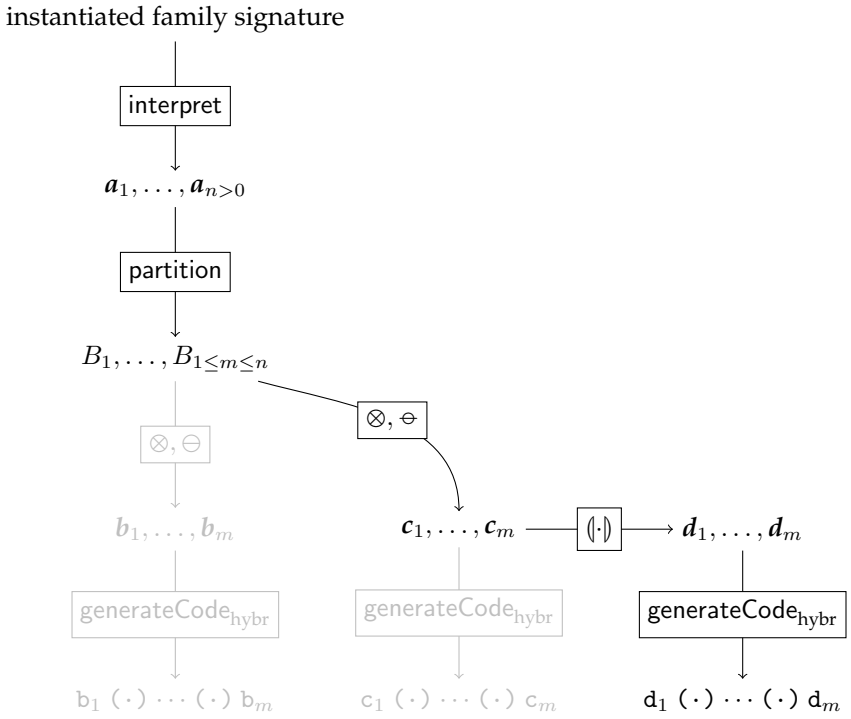


Figure 7.1: Hybrid compilation approach with syntactic subtraction and commandification

7.1 Theory

*(With Arbab, I previously published fragments of the material in this section in a conference paper [**J**A15b].)*

Data Commands

At run-time, general-purpose techniques for constraint solving—an NP-complete problem for finite domains—inflict not only overhead proportional to the size of a data constraint but also a constant overhead for preparing, making, and processing the result of the constraint solver call. Although one generally cannot escape using such techniques for checking arbitrary data constraints, a better alternative exists for many data constraints in practice. It starts with the observation that the data constraints in *all* constraint automata that I know of in the literature really constitute declarative specifications of a relatively straightforward imperative program. In this section, I therefore develop a technique for statically translating such a data constraint ϕ, off-line at compile-time, into a *data command*: a little imperative program that computes a data assignment σ

such that $\sigma \models \phi$, without general-purpose constraint solving. I call this kind of translation *commandification*. Essentially, I formalize and automate what programmers do when they write an imperative implementation of a declarative specification expressed as a data constraint. Figure 7.1 shows the resulting compilation approach, where $(\!|\cdot|\!)$ denotes commandification. By the end of this section, I make the class of data constraints currently supported by commandification precise.

I start by defining data commands, their semantics, and a proof system for reasoning about their execution.

Definition 44 (data commands)**.** *A data command is an object generated by the following grammar:*

$$\pi ::= \texttt{skip} \mid x := t \mid \phi \texttt{ -> } \pi \mid \pi \texttt{ ; } \pi \mid \varepsilon \qquad \textit{(data commands)}$$

$\mathbb{C}$OMM *denotes the set of all data commands.*

In the previous definition, ε denotes the empty data command, $x := t$ denotes an *assignment*, and ϕ -> π denotes a *failure statement*. Henceforth, I often write "value of x" instead of "the datum assigned to x".

I define an operational semantics for data commands based on an operational semantics for a sequential language by Apt et al. [AdBO09]. True to the idea that data commands solve data constraints, I model the *data state* that a data command executes in with either a function from data variables to data—a data assignment—or the distinguished object fail, which models abnormal termination. A *data configuration*, then, consists of a data command and a data state to execute that data command in.

Definition 45 (abnormal termination)**.** fail *is an unstructured object such that* fail $\notin \mathbb{A}$SSIGNM*.*

Definition 46 (data configurations)**.** *A data configuration is a pair* (π, ς) *where:*

- $\pi \in \mathbb{C}$OMM *(data command)*
- $\varsigma \in \mathbb{A}$SSIGNM $\cup \{\mathsf{fail}\}$ *(data state)*

$\mathbb{C}$ONF *denotes the set of all data configurations.*

A *transition system* on configurations formalizes their evolution in time.

Definition 47 (transition system on data configurations)**.** $\Longrightarrow \subseteq \mathbb{C}$ONF $\times$ $\mathbb{C}$ONF *denotes the smallest relation induced by the rules in Figure 7.2.*

Note that ϕ -> π indeed denotes a failure statement rather than a *conditional statement*: if the current data state violates the *guard* ϕ, execution abnormally terminates.

Through the transition system in Definition 47, I associate two different semantics with data commands. The *partial correctness semantics* of a data command π under a set of *initial* data state Σ consists of all the *final* data states Σ' to

$$\frac{}{(\texttt{skip}, \sigma) \Longrightarrow (\varepsilon, \sigma)} \tag{7.1}$$

$$\frac{}{(x := t, \sigma) \Longrightarrow (\varepsilon, \sigma[x \mapsto \mathsf{eval}_\sigma(t)])} \tag{7.2}$$

$$\frac{\sigma \models \phi}{(\phi \texttt{ -> } \pi, \sigma) \Longrightarrow (\pi, \sigma)} \tag{7.3} \qquad \frac{\sigma \not\models \phi}{(\phi \texttt{ -> } \pi, \sigma) \Longrightarrow (\varepsilon, \mathsf{fail})} \tag{7.4}$$

$$\frac{(\pi, \sigma) \Longrightarrow (\pi', \sigma') \textbf{ and } \pi' \neq \varepsilon}{(\pi \texttt{ ; } \pi'', \sigma) \Longrightarrow (\pi' \texttt{ ; } \pi'', \sigma')} \tag{7.5} \qquad \frac{(\pi, \sigma) \Longrightarrow (\varepsilon, \sigma')}{(\pi \texttt{ ; } \pi'', \sigma) \Longrightarrow (\pi'', \sigma')} \tag{7.6}$$

Figure 7.2: Addendum to Definition 47

which any of those initial states may evolve through execution of π. Notably, this partial correctness semantics ignores abnormal termination. In contrast, the *total correctness semantics* of π under Σ consists not only of Σ' but, if at least one execution abnormally terminates, also of fail.

Definition 48 (correctness semantics of data commands)**.** $\mathsf{Final}, \mathsf{Final}_{\mathsf{fail}} : \mathbb{C}\textsc{omm} \times 2^{\mathbb{A}\textsc{ssignm}} \to 2^{\mathbb{A}\textsc{ssignm} \cup \{\mathsf{fail}\}}$ *denote the functions defined by the following equations:*

$$\begin{aligned} \mathsf{Final}(\pi, \Sigma) &= \{\sigma' \mid \sigma \in \Sigma \textbf{ and } (\pi, \sigma) \Longrightarrow^* (\varepsilon, \sigma')\} \\ \mathsf{Final}_{\mathsf{fail}}(\pi, \Sigma) &= \mathsf{Final}(\pi, \Sigma) \cup \{\mathsf{fail} \mid \sigma \in \Sigma \textbf{ and } (\pi, \sigma) \Longrightarrow^* (\pi', \mathsf{fail})\} \end{aligned}$$

Apt et al. showed that all programs from a superset of the set of all data commands execute deterministically [AdBO09]. Consequently, also data commands execute deterministically.

Lemma 17 ([AdBO09])**.** $|\mathsf{Final}(\pi, \{\sigma\})| \leq 1 \textbf{ and } |\mathsf{Final}_{\mathsf{fail}}(\pi, \{\sigma\})| = 1$

To prove the correctness of commandification, I use Hoare logic [Hoa69], where *triples* $\{\phi\}\ \pi\ \{\phi'\}$ play a central role. In such a triple, *precondition* ϕ characterizes the set of initial data states, π denotes the data command to execute on those states, and *postcondition* ϕ' characterizes the set of final data states after executing π.

Definition 49 (triples)**.** $\mathbb{T}\textsc{ripl} = \mathbb{DC} \times \mathbb{C}\textsc{omm} \times \mathbb{DC}$ *denotes the set of all triples, typically denoted by* $\{\phi\}\ \pi\ \{\phi'\}$.

Let $[\![\phi]\!]$ denote the set of data states that satisfy ϕ (i.e., the data assignments characterized by ϕ). I interpret triples in two senses: that of partial correctness and that of total correctness. In the former case, a triple $\{\phi\}\ \pi\ \{\phi'\}$ holds true iff every final data state to which an initial data state characterized by ϕ can

$$\frac{}{\vdash_{\text{part}} \{\phi\}\ \texttt{skip}\ \{\phi\}} \quad (7.9) \qquad \frac{}{\vdash_{\text{tot}} \{\phi\}\ \texttt{skip}\ \{\phi\}} \quad (7.10)$$

$$\frac{}{\vdash_{\text{part}} \{\phi[t/x]\}\ x := t\ \{\phi\}} \quad (7.11) \qquad \frac{}{\vdash_{\text{tot}} \{\phi[t/x]\}\ x := t\ \{\phi\}} \quad (7.12)$$

$$\frac{\vdash_{\text{part}} \{\phi_1\}\ \pi_1\ \{\phi\} \quad \textbf{and} \quad \vdash_{\text{part}} \{\phi\}\ \pi_2\ \{\phi_2\}}{\vdash_{\text{part}} \{\phi_1\}\ \pi_1\ ;\ \pi_2\ \{\phi_2\}} \quad (7.13) \qquad \frac{\vdash_{\text{tot}} \{\phi_1\}\ \pi_1\ \{\phi\} \quad \textbf{and} \quad \vdash_{\text{tot}} \{\phi\}\ \pi_2\ \{\phi_2\}}{\vdash_{\text{tot}} \{\phi_1\}\ \pi_1\ ;\ \pi_2\ \{\phi_2\}} \quad (7.14)$$

$$\frac{\vdash_{\text{part}} \{\phi_1'\}\ \pi\ \{\phi_2'\} \quad \textbf{and} \quad \phi_1 \Rightarrow \phi_1' \quad \textbf{and} \quad \phi_2' \Rightarrow \phi_2}{\vdash_{\text{part}} \{\phi_1\}\ \pi\ \{\phi_2\}} \quad (7.15) \qquad \frac{\vdash_{\text{tot}} \{\phi_1'\}\ \pi\ \{\phi_2'\} \quad \textbf{and} \quad \phi_1 \Rightarrow \phi_1' \quad \textbf{and} \quad \phi_2' \Rightarrow \phi_2}{\vdash_{\text{tot}} \{\phi_1\}\ \pi\ \{\phi_2\}} \quad (7.16)$$

$$\frac{\vdash_{\text{part}} \{\phi \wedge \ell\}\ \pi\ \{\phi'\}}{\vdash_{\text{part}} \{\phi\}\ \ell\ \texttt{->}\ P\ \{\phi'\}} \quad (7.17) \qquad \frac{\vdash_{\text{tot}} \{\phi\}\ \pi\ \{\phi'\} \quad \textbf{and} \quad \phi \Rightarrow \ell}{\vdash_{\text{tot}} \{\phi\}\ \ell\ \texttt{->}\ \pi\ \{\phi'\}} \quad (7.18)$$

$$\frac{\vdash_{\text{part}} \{\phi\}\ \pi\ \{\phi_1\} \quad \textbf{and} \quad \vdash_{\text{tot}} \{\phi\}\ \pi\ \{\phi_2\}}{\vdash_{\text{tot}} \{\phi\}\ \pi\ \{\phi_1 \wedge \phi_2\}} \quad (7.19)$$

Figure 7.3: Addendum to Definition 51

evolve under π satisfies ϕ'; in the latter case, additionally, execution of π does not abnormally terminate.

Definition 50 (interpretation of triples). $\models_{\text{part}}, \models_{\text{tot}} \subseteq \mathbb{T}\text{RIPL}$ *denote the smallest relations induced by the following rules:*

$$\frac{\mathsf{Final}(\pi, [\![\phi]\!]) \subseteq [\![\phi']\!]}{\models_{\text{part}} \{\phi\}\ \pi\ \{\phi'\}} \quad (7.7) \qquad \frac{\mathsf{Final}_{\text{fail}}(\pi, [\![\phi]\!]) \subseteq [\![\phi']\!]}{\models_{\text{tot}} \{\phi\}\ \pi\ \{\phi'\}} \quad (7.8)$$

To prove properties of data commands, I use the following sound *proof systems* for partial and total correctness, adopted from Apt et al. with some minor cosmetic changes [AdBO09].

Definition 51 (proof systems of triples). $\vdash_{\text{part}}, \vdash_{\text{tot}} \subseteq \mathbb{T}\text{RIPL}$ *denote the smallest relations induced by the rules in Figure 7.3.*

Theorem 16 ([AdBO09]). $\vdash_{\text{part}} \{\phi\}\ \pi\ \{\phi'\}$ **implies** $\models_{\text{part}} \{\phi\}\ \pi\ \{\phi'\}$

Theorem 17 ([AdBO09]). $\vdash_{\text{tot}} \{\phi\}\ \pi\ \{\phi'\}$ **implies** $\models_{\text{tot}} \{\phi\}\ \pi\ \{\phi'\}$

Note that the first four rules for $\vdash_{\text{part}}$ and the first four rules for $\vdash_{\text{tot}}$ have the same premise/consequence. I use $\vdash_{\text{part}}$ to prove the *soundness* of commandification; I use $\vdash_{\text{tot}}$ to prove commandification's *completeness*.

Commandification

At run-time, to check if a transition (q, P, ϕ, q') can fire, a protocol unit first checks every port in P for that port's *readiness*. For instance, every input port should have a pending `put` (where "input" qualifies that port from the protocol perspective). Subsequently, the protocol unit checks whether a data state σ exists that (i) satisfies ϕ and (ii) subsumes an *initial data state* σ_{init} (i.e., $\sigma_{\text{init}} \subseteq \sigma$). If so, I call σ a *solution* of ϕ under σ_{init}. The domain of σ_{init} contains all *uncontrollable data variables* in ϕ: the input ports in P (intersected with $\mathsf{Free}(\phi)$) and ${}^\bullet m$ for every memory cell m in the constraint automaton (also intersected with $\mathsf{Free}(\phi)$). More precisely, σ_{init} maps every input port p in $\mathsf{Free}(\phi)$ to the particular datum *forced* to pass through p by the unit of parallelism on p's other side (e.g., the datum involved in p's pending `put`, performed by a neighboring worker unit), while σ_{init} maps every ${}^\bullet m$ in $\mathsf{Free}(\phi)$ to the datum that currently resides in m. Thus, before the protocol unit invokes a constraint solver for ϕ, it already fixes values for all uncontrollable data variables in ϕ; when subsequently invoked, a constraint solver may, in search of a solution for ϕ under σ_{init}, select values only for data variables outside σ_{init}'s domain. Slightly more formally:

$$\sigma_{\text{init}} = \left\{ p \mapsto d \;\middle|\; \begin{array}{l}[\text{the } \texttt{put} \text{ pending on input port } p \text{ involves datum } d] \\ \textbf{and}\ \ p \in \mathsf{Free}(\phi)\end{array} \right\} \cup \left\{ {}^\bullet m \mapsto d \;\middle|\; \begin{array}{l}[\text{memory cell } m \text{ currently contains datum } d] \\ \textbf{and}\ \ {}^\bullet m \in \mathsf{Free}(\phi)\end{array} \right\}$$

With commandification, instead of invoking a constraint solver, the protocol unit executes a compiler-generated data command for ϕ on σ_{init}, thereby gradually extending σ_{init} to a full solution. This compiler-generated data command essentially works as an efficient, little, dedicated constraint solver for ϕ.

To commandify a data constraint of the form $\ell_1 \wedge \cdots \wedge \ell_k$, I construct a data command that (i) enforces as many data literals of the form $t_1 = t_2$ as possible with assignments and (ii) checks all remaining data literals with failure statements. I call data literals of the form $t_1 = t_2$ *data equalities*. To exemplify such commandification, recall data constraint ϕ_{eg} on page 178. In this data constraint, let C denote an input port and let x denote a memory cell. In that case, the set of uncontrollable data variables in ϕ_{eg} consists of C and ${}^\bullet$x. Now, ϕ_{eg}

has six correct commandifications:

$\pi_1 =$ B := •x ;
D := C ;
E := add(B, D) ;
F := E ;
G := E
¬Odd(G) -> skip ;

$\pi_2 =$ B := •x ;
D := C ;
E := add(B, D) ;
G := E
F := E ;
¬Odd(G) -> skip ;

$\pi_3 =$ B := •x ;
D := C ;
E := add(B, D) ;
G := E
¬Odd(G) -> skip ;
F := E ;

$\pi_4 =$ D := C ;
B := •x ;
E := add(B, D) ;
F := E ;
G := E
¬Odd(G) -> skip ;

$\pi_5 =$ D := C ;
B := •x ;
E := add(B, D) ;
G := E
F := E ;
¬Odd(G) -> skip ;

$\pi_6 =$ D := C ;
B := •x ;
E := add(B, D) ;
G := E
¬Odd(G) -> skip ;
F := E ;

I stipulate the same precondition for each of these data commands, namely that •x and C have a non-nil value (later formalized as data literals •x = •x and C = C). This precondition models that the execution of these data commands should always start on an initial data state over the uncontrollable data variables •x and C. Under this precondition, if a protocol unit executes π_1, it first assigns the values of •x and C to B and D. Subsequently, it assigns the evaluation of add(B, D) to E. Next, it assigns the value of E to F and G. Finally, it checks ¬Odd(G) with a failure statement. Data commands π_2 and π_3 differ from data command π_1 only in the order of the last three steps; data commands π_4, π_5 and π_6 differ from π_1, π_2 and π_3 only in the order of the first two steps. If execution of π_i on σ_{init} successfully terminates, the resulting final data state σ satisfies ϕ_{eg}. I call this *soundness*. Moreover, if a σ' exists such that $\sigma' \models \phi_{\text{eg}}$ and $\sigma_{\text{init}} \subseteq \sigma'$, execution of π_i successfully terminates. I call this *completeness*.

Generally, soundness and completeness crucially depend on the order in which assignments and failure statements follow each other in π. For instance, changing the order of G := E and ¬Odd(G) -> skip in the previous example yields a data command whose execution always fails (because G does not have a value yet on evaluating the guard of the failure statement). Such a trivially sound but incomplete data constraint serves no use. As another complication, not every data equality can become an assignment. In a first class of cases, neither the left-hand side nor the right-hand side of a data equality matches data variable x. For instance, I *must* translate add(B, D) = mult(B, D) into a failure statement, because I clearly cannot assign either of its two operands to the other. In a second class of cases, multiple data equalities in a data constraint have a left-hand side or a right-hand side that matches the same data variable x. For instance, I can translate only one data equality in E = add(B, D) ∧ E = mult(B, D) into an assignment, after which I *must* translate the other one into a failure statement, to avoid conflicting assignments to E.

To deal with these complications, I define a *precedence relation* on the data literals in a normal data constraint that formalizes their dependencies. Re-

$$\frac{x = t, \ell \in \mathsf{Liter}^{=}(\varphi) \ \textbf{and} \ x \in \mathsf{Variabl}(\ell)}{x = t \sqsubseteq \ell} \tag{7.20}$$

$$\frac{x = t, \ell \in \mathsf{Liter}^{=}(\varphi) \ \textbf{and} \ \left[\ell \neq x' = t' \ \textbf{for all} \ x', t'\right]}{x = t \sqsubseteq \ell} \tag{7.21}$$

$$\frac{\ell_1 \sqsubseteq \ell_2 \ \textbf{and} \ \ell_2 \sqsubseteq \ell_3 \ \textbf{and} \ \ell_2 \notin \{\ell_1, \ell_3\}}{\ell_1 \sqsubseteq \ell_3} \tag{7.22}$$

Figure 7.4: Addendum to Definition 53

call from Definition 38 that every normal data constraint consists of a conjunctive kernel of data literals, enveloped with existential quantifications. First, for technical convenience, I introduce a function that extends $\mathsf{Liter}(\varphi)$ (i.e., the data literals in the kernel of φ) with its "symmetric data equalities".

Definition 52 (=-symmetric closure)**.** $\mathsf{Liter}^{=} : \mathbb{DC}_{\exists,\wedge} \to 2^{\mathbb{DC}_{\exists,\wedge}}$ *denotes the function defined by the following equation:*

$$\mathsf{Liter}^{=}(\varphi) = \mathsf{Liter}(\varphi) \cup \{t_2 = t_1 \mid t_1 = t_2 \in \mathsf{Liter}(\varphi)\}$$

Obviously, because $t_1 = t_2 \equiv t_2 = t_1$, I have $\bigwedge \mathsf{Liter}(\varphi) \equiv \bigwedge \mathsf{Liter}^{=}(\varphi)$ for all φ.

Definition 53 (precedence I)**.** $\sqsubseteq : \mathbb{DC}_{\exists,\wedge} \to 2^{\mathbb{DC} \times \mathbb{DC}}$ *denotes the function defined by the following equation:*

$$\sqsubseteq(\varphi) = \sqsubseteq$$

where $\sqsubseteq$ *denotes the smallest relation induced by the rules in Figure 7.4.*

I usually write $\sqsubseteq_\varphi$ instead of $\sqsubseteq(\varphi)$ and use $\sqsubseteq_\varphi$ as an infix relation. In words, $x = t \sqsubseteq_\varphi \ell$ means that the assignment $x := t$ precedes the commandification of ℓ (i.e., ℓ depends on x). Rule 7.20 deals with the previously discussed first class of data-equalities-that-cannot-become-assignments, by imposing precedence only on data literals of the form $x = t$; shortly, I comment on the second class of data-equalities-that-cannot-become-assignments. Rule 7.21 conveniently ensures that every $x = t$ precedes all differently shaped data literals. Strictly speaking, I probably do not need this rule, but it simplifies some notation and proofs later on.

For the sake of argument—generally, this does *not* hold true—suppose that a precedence relation $\sqsubseteq_\varphi$ denotes a *strict partial order* on $\mathsf{Liter}^{=}(\varphi)$. In that case, I can *linearize* $\sqsubseteq_\varphi$ to a strict total order $<$ (i.e., embedding $\sqsubseteq_\varphi$ into $<$ such that $\sqsubseteq_\varphi \subseteq <$) with a topological sort on the digraph $(\mathsf{Liter}^{=}(\varphi), \sqsubseteq_\varphi)$ [Kah62, Knu97]. Intuitively, such a linearization gives me an order in which I can translate data literals in $\mathsf{Liter}^{=}(\varphi)$ to data commands in a sound and complete way.

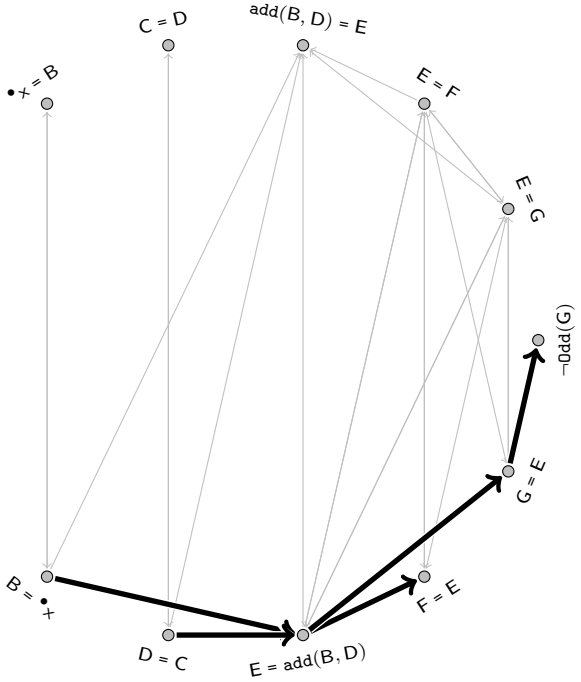


Figure 7.5: Digraph for precedence relation $\sqsubseteq_{\phi_{\text{eg}}}$ (without loop arcs and without arcs induced by Rule 7.21, to avoid further clutter). An arc (ℓ, ℓ') corresponds to $\ell \sqsubseteq_{\phi_{\text{eg}}} \ell'$. Arcs between the same data vertices, but in different directions, lie on top of each other. Bold arcs represent a fragment of the strict partial order extracted from $\sqsubseteq_{\phi_{\text{eg}}}$.

Shortly, I give an algorithm for doing so and indeed prove its correctness. Problematically, however, $\sqsubseteq_\varphi$ generally does not denote a strict partial order: generally, it violates asymmetry and irreflexivity (i.e., graph-theoretically, it contains many cycles). For instance, Figure 7.5 shows the digraph $(\mathsf{Liter}^{=}(\phi_{\text{eg}}), \sqsubseteq_{\phi_{\text{eg}}})$, which indeed contains cycles. For now, I defer this issue to the next subsection, because it forms a concern orthogonal to the commandification algorithm and its correctness. Until then, I simply assume the existence of a procedure for extracting a strict partial order from $\sqsubseteq_\varphi$, represented by bold arcs in Figure 7.5.

Algorithm 2 translates a normal data constraint φ, a set of variables X, and a binary relation on data literals $<$ to a data command π. It requires the following on its input. First, $<$ should denote a strict total order on the =-symmetric closure of φ's data literals. Let n denote *a*—not necessarily *the*—number of data equalities in $\mathsf{Liter}^{=}(\varphi)$, and let m denote the number of remaining data literals in $\mathsf{Liter}^{=}(\varphi)$. Then, $\ell_1, \ldots, \ell_{n+m}$ denote the data literals in $\mathsf{Liter}^{=}(\varphi)$ such that (i) their indices respect $<$ and (ii) every ℓ_i denotes $x_i = t_i$ for $1 \leq i \leq n$. Next, for every data variable in a data literal in $\mathsf{Liter}^{=}(\varphi)$, but outside the set

Algorithm 2 Algorithm for translating a normal data constraint φ, a set of variables X, and a binary relation on data literals $<$ to a data command π

Require: $<$ denotes a strict total order on $\mathsf{Liter}^{=}(\varphi)$
and $\mathsf{Liter}^{=}(\varphi) = \{\ell_1, \ldots, \ell_{n+m}\}$
and $\ell_1 < \cdots < \ell_n < \ell_{n+1} < \cdots < \ell_{n+m}$
and $\ell_1 = x_1 = t_1$ **and** $\cdots$ **and** $\ell_n = x_n = t_n$
and $\mathsf{Variabl}(\varphi) \setminus X \subseteq \{x_1, \ldots, x_n\}$
and
$$\left[\begin{array}{c}\left[\begin{array}{l}\left[x = t \in \mathsf{Liter}^{=}(\varphi) \text{ and } x' \in \mathsf{Variabl}(t)\right] \text{ implies}\\ \quad \left[x' \in X \text{ or } \left[x' = t' < x = t \text{ for some } t'\right]\right]\end{array}\right]\\ \text{for all } x, x', t\end{array}\right]$$

function ALGORITHM2($\varphi, X, <$)
$\pi := \texttt{skip}$
$i := 1$
while $i \leq n$ **do**
if $x_i \in X \cup \{x_1, \ldots, x_{i-1}\}$ **then**
$\pi := (\pi \texttt{ ; } x_i = t_i \texttt{ -> skip})$
else
$\pi := (\pi \texttt{ ; } x_i := t_i)$
$i := i + 1$
while $i \leq n + m$ **do**
$\pi := (\pi \texttt{ ; } \ell_i \texttt{ -> skip})$
$i := i + 1$
return π

Ensure: $\vdash_{\text{part}} \{\bigwedge\{x = x \mid x \in X\}\}\ \pi\ \{\ell_1 \wedge \cdots \wedge \ell_{n+m}\}$
and
$$\left[\left[\begin{array}{l}\sigma \models \ell_1 \wedge \cdots \wedge \ell_{n+m} \text{ implies}\\ \vdash_{\text{tot}} \{\bigwedge\{x = \sigma(x) \mid x \in X\}\}\\ \quad \pi\\ \quad \{\bigwedge\{x = \sigma(x) \mid x \in X \cup \{x_1, \ldots, x_n\}\}\}\end{array}\right] \text{ for all } \sigma\right]$$

of uncontrollable variables X, a data equality $x_i = t_i$ should exist. Otherwise, such a data variable can get a value only through search—exactly what commandification tries to avoid—and not through assignment; *underspecified* data constraints fundamentally lie outside the scope of commandification in general and Algorithm 2 in particular. Finally, if term t in a data equality $x = t$ depends on a variable x', a data equality $x' = t'$ should precede $x = t$ under $<$. The rules in Definition 53 induce precedence relations for which all these requirements hold true, except that those precedence relations not necessarily denote strict partial orders and, hence, may not admit linearization. Consequently, the precedence relations in Definition 53 may not yield strict total orders as required by Algorithm 2. I address this issue in the next subsection.

Assuming satisfaction of its requirements, Algorithm 2 works as follows. It first loops over the first n (according to $<$) x_i = t_i data literals. If an assignment for x_i already exists in the data command under construction π, Algorithm 2 translates x_i = t_i to a failure statement; otherwise, it translates x_i = t_i to an assignment. This approach resolves issues with the previously discussed second class of equalities-that-cannot-become-assignments. After the first loop, the algorithm uses a second loop to translate the remaining m data literals to failure statements. The algorithm runs in time linear in $n + m$, and it terminates.

Algorithm 2 ensures the soundness and completeness of π. Note that I use a different proof system for soundness (partial correctness, $\vdash_{\text{part}}$) than for completeness (total correctness, $\vdash_{\text{tot}}$).

Theorem 18. *Algorithm 2 is correct.*

Algorithm 2 has the minor issue that it may produce more failure statements than strictly necessary. For instance, if I run Algorithm 2 on the total order extracted from $\sqsubseteq_{\phi_{\text{eg}}}$ in Figure 7.5, I get both the assignment `D := C` and the unnecessary failure statement `C = D -> skip`. After all, the digraph contains both `D = C` and `C = D`, one of which I added while computing $\mathsf{Liter}^{=}(\phi_{\text{eg}})$ to account for the symmetry of `=`. Generally, such symmetric data literals result either in one assignment and one failure statement or in two failure statements; one can easily prove that symmetric data literals never result in two assignments. In both cases, one can safely remove one of the failure statements, because successful termination of the remaining statement already accounts for the removed failure statement.

Commandification with Cycles

Algorithm 2 requires that $<$ denotes a strict total order. Precedence relations in Definition 53 of $\sqsubseteq$, however, do not yield such orders: graph-theoretically, they may contain cycles. In this subsection, I present a solution for this problem. I start by extending the previous precedence relations with a unique least element, denoted by $\bigstar$, and by making dependencies of data literals on uncontrollable variables explicit. In the following definition, let X denote such a set of uncontrollable variables.

Definition 54 (precedence II)**.** $\sqsubseteq : \mathbb{DC}_{\exists,\wedge} \times 2^{\mathbb{X}} \to 2^{(\mathbb{DC} \cup \{\bigstar\}) \times \mathbb{DC}}$ *denotes the function defined by the following equation:*

$$\sqsubseteq(\varphi, X) = \sqsubseteq$$

where $\sqsubseteq$ *denotes the smallest relation induced by the rules in Figure 7.6.*

I usually write $\sqsubseteq_{\varphi}^{X}$ instead of $\sqsubseteq(\varphi, X)$ and use $\sqsubseteq_{\varphi}^{X}$ as an infix relation. The two new rules state that data literals in which only uncontrollable variables occur "depend" on $\bigstar$.

$$\frac{\ell_1 \sqsubseteq_\phi \ell_2}{\ell_1 \sqsubseteq \ell_2} \tag{7.23}$$

$$\frac{\ell \in \mathsf{Liter}^{=}(\varphi) \quad \textbf{and} \quad \mathsf{Variabl}(\ell) \subseteq X}{\bigstar \sqsubseteq \ell} \tag{7.24}$$

$$\frac{x = t \in \mathsf{Liter}^{=} \quad \textbf{and} \quad \mathsf{Variabl}(t) \subseteq X}{\bigstar \sqsubseteq x = t} \tag{7.25}$$

Figure 7.6: Addendum to Definition 54

A precedence relation $\sqsubseteq_\varphi^X$ denotes a strict partial order if its corresponding digraph $(\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\}, \sqsubseteq_\varphi^X)$ defines a $\bigstar$-*arborescence*: a digraph consisting of $n-1$ arcs such that a path exists from $\bigstar$ to each of its n vertices [KV08]. Equivalently, in a $\bigstar$-arborescence, $\bigstar$ has no incoming arcs, every other vertex has exactly one incoming arc, and the arcs form no cycles [KV08]. The first formulation seems more intuitive here: every path from $\bigstar$ to some data literal ℓ represents an order in which Algorithm 2 should translate the data literals on that path to ensure the correctness of the translation of ℓ. The second formulation simplifies observing that arborescences correspond to strict partial orders.

A naive approach to extract a strict partial order from $\sqsubseteq_\varphi^X$ consists of computing a $\bigstar$-arborescence of the digraph $(\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\}, \sqsubseteq_\varphi^X)$. Even if such a $\bigstar$-arborescence exists, however, this approach does not work as expected if $\mathsf{Liter}^{=}(\varphi)$ contains a data literal $x = t$ where t has more than one data variable. For instance, by definition, every arborescence of the digraph in Figure 7.5 has only one incoming arc for $\mathsf{E} = \mathtt{add}(\mathsf{B}, \mathsf{D})$, even though assignments to *both* B *and* D must precede an assignment to E. Because these dependencies exist as two separate arcs, no arborescence can capture them. To solve this, I should somehow represent the dependencies of $\mathsf{E} = \mathtt{add}(\mathsf{B}, \mathsf{D})$ with a single incoming arc. I can do so by allowing arcs to have multiple tails (i.e., one for every data variable). In that case, I can replace the two separate incoming arcs of $\mathsf{E} = \mathtt{add}(\mathsf{B}, \mathsf{D})$ with a single two-tailed incoming arc as in Figure 7.7. The two tails make explicit that to evaluate `add`, I need values for both its arguments: multiple tails represent a conjunction of dependencies of a data literal.

By combining single-tailed arcs into multiple-tailed arcs, I effectively transform the digraphs considered so far into B-*graphs*, a special kind of hypergraph with only B-*arcs* (i.e., *b*ackward hyperarcs, i.e., hyperarcs with exactly one head) [GLPN93]. Generally, I cannot derive such B-graphs from precedence relations as in Definition 54: their richer structure makes B-graphs more expressive—they convey strictly more information—than digraphs. In contrast, I can easily transform a B-graph into a precedence relation by splitting B-arcs into single-tailed arcs in the obvious way. Deriving precedence relations from more expressive B-graphs therefore constitutes a correct way of obtaining strict total orders that satisfy the requirements of Algorithm 2; doing so just

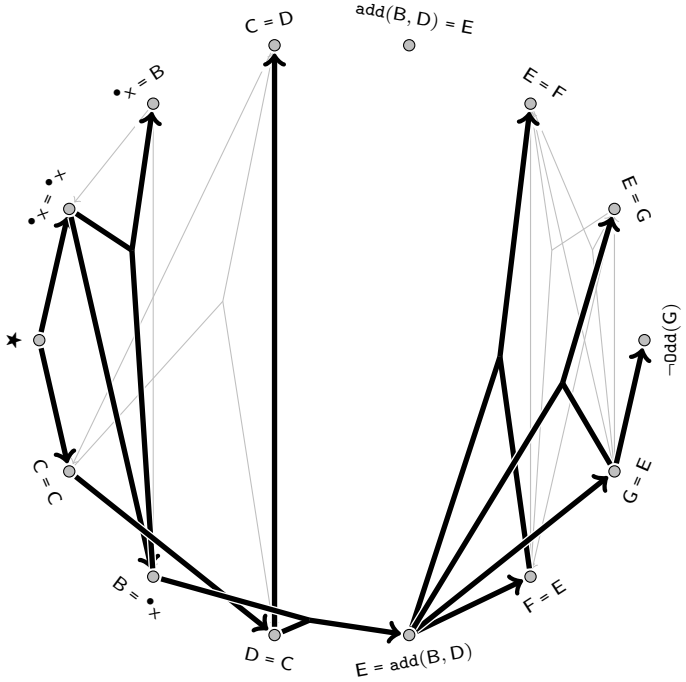


Figure 7.7: B-graph corresponding to the digraph in Figure 7.5 (without loop B-arcs and without three-tailed B-arcs, to avoid further clutter). An arc (ℓ, ℓ') corresponds to $\ell \sqsubseteq_{\phi_{\text{eg}}} \ell'$. Bold arcs represent an arborescence.

eliminates information that this algorithm does not care about.

Thus, I propose the following. Instead of formalizing dependencies among data literals in a set $\text{Liter}^{=}(\varphi) \cup \{\bigstar\}$ directly as a precedence relation, I first formalize those dependencies as a B-graph. If the resulting B-graph defines a $\bigstar$-arborescence, I can directly extract a cycle-free precedence relation $\sqsubset$. Otherwise, I compute a $\bigstar$-arborescence of the resulting B-graph and extract a cycle-free precedence relation $\sqsubset$ afterward. Either way, $\sqsubset$ denotes a strict partial order whose linearization satisfies the requirements in Algorithm 2.

Definition 55 (B-precedence)**.** $\blacktriangleleft : \mathbb{DC}_{\exists,\wedge} \times 2^{\mathbb{X}} \to 2^{(2^{\mathbb{DC}} \cup \{\bigstar\}) \times \mathbb{DC}}$ *denotes the function defined by the following equation:*

$$\blacktriangleleft(\varphi, X) = \blacktriangleleft$$

where $\blacktriangleleft$ *denotes the smallest relation induced by the rules in Figure 7.8.*

I usually write $\blacktriangleleft_\varphi^X$ instead of $\blacktriangleleft(\varphi, X)$ and use $\blacktriangleleft_\varphi^X$ as an infix relation. Rule 7.26 generalizes Rule 7.20 in Definition 53, by joining sets of dependencies of a data literal in a single B-arc. Rule 7.27 states that $x = t$ does not necessarily depend

$$\frac{\begin{array}{l}\ell \in \mathsf{Liter}^{=}(\varphi) \\ \textbf{and } \mathsf{Variabl}(\ell) = \{x_1, \ldots, x_k\} \\ \textbf{and } x_1 = t_1, \ldots, x_k = t_k \in \mathsf{Liter}^{=}(\varphi) \cup \{\hat{x} = \hat{x} \mid \hat{x} \in X\}\end{array}}{\{x_1 = t_1, \ldots, x_k = t_k\} \blacktriangleleft \ell} \tag{7.26}$$

$$\frac{\begin{array}{l}x = t \in \mathsf{Liter}^{=}(\varphi) \\ \textbf{and } \mathsf{Variabl}(t) = \{x_1, \ldots, x_k\} \\ \textbf{and } x_1 = t_1, \ldots, x_k = t_k \in \mathsf{Liter}^{=}(\varphi) \cup \{\hat{x} = \hat{x} \mid \hat{x} \in X\}\end{array}}{\{x_1 = t_1, \ldots, x_k = t_k\} \blacktriangleleft x = t} \tag{7.27}$$

$$\frac{x \in X}{\bigstar \blacktriangleleft x = x} \tag{7.28}$$

Figure 7.8: Addendum to Definition 55

on x—as implied by Rule 7.26—but only on the free variables in t (i.e., I can derive a value for x from values of the data variables in t). Note that through Rules 7.26 and 7.27, I extend the previous domain $\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\}$ with *semantically insignificant* data equalities of the form $x = x$, each of which I relate to $\bigstar$ with Rule 7.28. I do this only for the technical convenience of treating both uncontrollable variables in X (which may have no data equalities in $\mathsf{Liter}^{=}(\varphi)$) and the other variables (which must have) in a uniform way. For instance, Figure 7.7 shows the B-graph for data constraint ϕ_{eg}.

Generally, in a B-graph, data literals can have multiple incoming B-arcs. Such multiple incoming B-arcs represent a disjunction of conjunctions of dependencies. Importantly, as long as Algorithm 2 respects the dependencies represented by *one* incoming B-arc, the other incoming B-arcs do not matter. An arborescence, which contains one incoming B-arc for every data literal, therefore preserves enough dependencies. Theorem 19 makes this more precise.

One can straightforwardly compute an arborescence of a B-graph

$$(\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\} \cup \{x = x \mid x \in X\}, \blacktriangleleft_{\varphi}^{X})$$

with an exploration algorithm reminiscent of breadth-first search. First, let $\lhd \subseteq \blacktriangleleft_{\varphi}^{X}$ denote the arborescence under computation, and let $L_{\text{done}} \subseteq \mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\} \cup \{x = x \mid x \in X\}$ denote the set of vertices (i.e., data literals) already explored; initially, $\lhd = \emptyset$ and $L_{\text{done}} = \{\bigstar\}$. Now, given some L_{done}, compute a set of vertices L_{next} connected only to vertices in L_{done} by a B-arc in $\blacktriangleleft_{\varphi}^{X}$. Then, for every vertex in L_{next}, add an incoming B-arc to $\lhd$.[1] Afterward, add L_{next} to L_{done}. Repeat this process until L_{next} becomes empty. Once that happens,

[1] If a vertex ℓ in L_{next} has multiple incoming B-arcs, the choice among them matters not: the choice remains local, because every B-arc has only one head (i.e., adding an ℓ-headed B-arc to $\lhd$ cannot cause another vertex to get multiple incoming B-arcs, which would invalidate the arbores-

$$\frac{\ell_1 \in \mathsf{Liter}^{=}(\varphi) \cap L \ \textbf{and} \ L \lhd_{\varphi}^{X} \ell_2}{\ell_1 \sqsubset \ell_2} \tag{7.29}$$

$$\frac{x = t, \ell \in \mathsf{Liter}^{=}(\varphi) \ \textbf{and} \ \left[\ell \neq x' = t' \ \textbf{for all} \ x', t'\right]}{x = t \sqsubset \ell} \tag{7.30}$$

$$\frac{\ell_1 \sqsubset \ell_2 \ \textbf{and} \ \ell_2 \sqsubset \ell_3 \ \textbf{and} \ \ell_2 \notin \{\ell_1, \ell_3\}}{\ell_1 \sqsubset \ell_3} \tag{7.31}$$

Figure 7.9: Addendum to Definition 56

either $\lhd$ contains an arborescence (if $L_{\text{done}} = L$) or no arborescence exists. This computation runs in linear time, in the size of the B-graph. See also Footnote 1. Henceforth, let $\lhd_{\varphi}^{X}$ denote the final arborescence so computed; if no arborescence exists, I stipulate $\lhd_{\varphi}^{X} = \emptyset$.

Definition 56 (precedence III). $\sqsubset : \mathbb{DC}_{\exists,\wedge} \times 2^{\mathbb{X}} \to \mathbb{DC} \times \mathbb{DC}$ *denotes the function defined by the following equation:*

$$\sqsubset(\varphi, X) = \sqsubset$$

where $\sqsubset$ denotes the smallest relation induced by the rules in Figure 7.9.

I usually write $\sqsubset_{\varphi}^{X}$ instead of $\sqsubset(\varphi, X)$. Rules 7.30 and 7.31 have the same premise/consequence as Rules 7.21 and 7.22; Rule 7.29 straightforwardly splits B-arcs into single-tailed arcs. For instance, the bold arcs in Figure 7.5 represent a fragment of the precedence relation so derived from the arborescence in Figure 7.7.

For every $\sqsubset_{\varphi}^{X}$ induced from a nonempty $\bigstar$-arborescence (i.e., $\lhd_{\varphi}^{X} \neq \emptyset$), let $<_{\varphi}^{X}$ denote its linearization. The following theorem states that this linearization satisfies the requirements of Algorithm 2.

Theorem 19. $\lhd_{\varphi}^{X} \neq \emptyset$ **implies** $\left[(\varphi, X, <_{\varphi}^{X})\ \textit{satisfies Algorithm 2}\right]$

If the B-graph $(\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\} \cup \{x = x \mid x \in X\}, \blacktriangleleft_{\varphi}^{X})$ neither defines nor contains a $\bigstar$-arborescence, no B-graph equivalent of a path [AFF01] exists from $\bigstar$ to at least one vertex ℓ. In that case, the other vertices fail to resolve at least one of ℓ's dependencies. This occurs, for instance, when ℓ depends on x, but the B-graph contains no $x = t$ vertex. As another example, consider a recursive data equality $x = t$ with $x \in \mathsf{Variabl}(t)$: unless another data equality $x = t'$

cence). General hypergraphs, whose hyperarcs can have multiple heads, violate this property (i.e., the choice of which hyperarc to add becomes global instead of local). As a result, and in stark contrast to B-graphs, one cannot compute arborescences of general hypergraphs—an NP-complete problem [Woe92]—in polynomial time (if P $\neq$ NP).

with $t \neq t'$ exists, all its incoming B-arcs contain loops to itself. Consequently, no arborescence exists. In practice, such cases inherently require constraint solving techniques with backtracking to find a value for x. Nonexistence of a ★-arborescence thus signals a fundamental limitation to the applicability of Algorithm 2 (although mixed techniques of translating some parts of a data constraint to a data command at compile-time and leaving other parts to a constraint solver at run-time seem worthwhile to explore; I leave those for future work). Thus, the set of data constraints to which I can apply Algorithm 2 contains those (i) whose B-graph has a ★-arborescence, which guarantees linearizability of the induced precedence relation, and (ii) that satisfy also the rest of the requirements in Algorithm 2.

The constraint programming community has already observed that, for constraint solving, "if domain specific methods are available they should be applied *instead* [sic] of the general methods" [Apt09a]. Commandification pushes this piece of conventional wisdom to an extreme: essentially, every data command generated for a data constraint ϕ by Algorithm 2 constitutes a little, dedicated constraint solver capable of solving only ϕ. Nevertheless, execution of data commands bears similarities with *constraint propagation* techniques, in particular with *forward checking* [BMFL02]. Generally, constraint propagation aims to reduce the search space of a constraint satisfaction problem by transforming it into an equivalent "simpler" one, where variables have smaller domains, or where constraints refer to fewer variables. With forward checking, whenever a variable x gets a value d, a constraint solver removes values from the domains of all subsequent variables that, given d, violate a constraint. In the case of an equality $x = x'$, for instance, forward checking reduces the domain of x' to the singleton $\{d\}$ after an assignment of d to x. Commandification implicitly uses that same property of equality, but instead of explicitly representing the domain of a variable and the reduction of this domain to a singleton at run-time, commandification already turns the equality into an assignment at compile-time. Commandification may also remind one of classical *Gaussian elimination* for solving systems of linear equations over the reals [Apt09b]: there too, one orders variables and substitutes values/expressions for variables in other expressions. Data constraints, however, have a significantly different structure from real numbers, which makes solving data constraints directly via Gaussian elimination at least not obvious.

Commandification in Constraint Automata

To formally introduce data commands in constraint automata, I introduce commandification as a unary operation on constraint automata. First, because I want to avoid ad-hoc modifications to Definitions 15 and 19 (of data constraints and constraint automata), I present an encoding of data commands as data relations. In the following definition, let φ denote a normal data constraint in a normalized constraint automaton, let X denote the set of uncontrollable data variables in φ, and let $x_1, \ldots, x_k$ denote the free data variables in φ, ordered by $<_{\mathbb{T}\text{ERM}}$. Then, the data relation R, which encodes the commandification π

of φ, holds true of a data tuple $(d_1, \ldots, d_k)$ iff execution of π on an initial data state (over the variables in X) successfully terminates on a data state σ that maps every x_i to d_i.

Definition 57 (data commands as data relations)**.** $\mathsf{comm} : \mathbb{DC}_{\exists,\wedge} \times 2^{\mathbb{X}} \to \mathbb{DC}_{\exists,\wedge}$ *denotes the function defined by the following equation:*

$$\mathsf{comm}(\varphi, X) = \begin{cases} R(x_1, \ldots, x_k) \ \textbf{if} \ \begin{bmatrix} \mathsf{Free}(\varphi) = \{x_1, \ldots, x_k\} \\ \textbf{and} \ x_1 <_{\mathbb{TERM}} \cdots <_{\mathbb{TERM}} x_k \\ \textbf{and} \ \lhd_\varphi^X \neq \emptyset \\ \textbf{and} \ X \subseteq \mathsf{Free}(\varphi) \end{bmatrix} \\ \varphi \qquad\qquad \textbf{otherwise} \end{cases}$$

where R denotes the smallest relation induced by the following rule:

$$\frac{\begin{array}{l} \pi = \textsc{Algorithm2}(\varphi, X, \sqsubset_\varphi^X) \\ \textbf{and} \ \sigma \in \mathsf{Final}(\pi, [\![\bigwedge\{x = x \mid x \in X\}]\!]) \\ \textbf{and} \ \sigma(x_1), \ldots, \sigma(x_k) \in \mathbb{D} \end{array}}{(\sigma(x_1), \ldots, \sigma(x_k)) \in R} \tag{7.32}$$

Note that σ in Rule 7.32 may map also data variables outside $\mathsf{Free}(\varphi)$. This happens, for instance, with data constraints with existential quantifiers. The data commands for such data constraints explicitly assign values to quantified data variables, even though those variables do not qualify as free. Because $\{x_1 \mapsto d_1, \ldots, x_k \mapsto d_k\}$ contains the free data variables in φ, however, the additional data variables mapped by σ cannot affect the truth of φ (i.e., generally, $\models$ satisfies monotonicity of a data constraint ϕ in data states whose domain contains at least the free data variables in ϕ).

I define commandification in constraint automata in terms of comm.

Definition 58 (commandification)**.** $(\!|\cdot|\!) : \mathbb{AUTOM} \to \mathbb{AUTOM}$ *denotes the function defined by the following equation:*

$$\begin{array}{l} (\!|(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0))|\!) = \\ \quad (Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, (\!|\longrightarrow|\!), (q^0, \mu^0)) \end{array}$$

where $(\!|\longrightarrow|\!)$ denotes the smallest relation induced by the following rules:

$$\frac{\begin{array}{l} q \xrightarrow{P,\phi} q' \ \textbf{and} \ \phi \in \mathbb{DC}_{\exists,\wedge} \\ \textbf{and} \ X^{\text{init}} = P^{\text{in}} \cup {}^\bullet M \end{array}}{q \ (\!|\xrightarrow{P,\mathsf{comm}(\phi,\mathsf{Free}(\phi)\cap X^{\text{init}})}|\!) \ q'} \tag{7.33}$$

$$\frac{q \xrightarrow{P,\phi} q' \ \textbf{and} \ \phi \notin \mathbb{DC}_{\exists,\wedge}}{q \ (\!|\xrightarrow{P,\phi}|\!) \ q'} \tag{7.34}$$

Before I can actually adopt the compilation approach in Figure 7.1 in practice, I must establish the correctness and effectiveness of commandification. I consider commandification correct if it yields a behaviorally congruent—hence, behaviorally equivalent by Theorem 1—constraint automaton to the

original one. Before formulating this property as a theorem, the following lemma states the equivalence of a data constraint and its commandification.

Lemma 18. $\varphi \equiv \mathsf{comm}(\varphi, X)$

From Lemma 18, I conclude the following correctness theorem.

Theorem 20. $\boldsymbol{a} \simeq (\!|\boldsymbol{a}|\!)$

Note that this theorem works not only for normalized constraint automata but also for arbitrary ones. Normalization plays a role only in the sequel, where I show that commandification has its intended effect when applied to normalized constraint automata.

I consider commandification effective if, after commandifying a normalized constraint automaton $|\boldsymbol{a}|$, every data constraint in this automaton either encodes a data command as in Definition 57 or has no data variables in it (in which case a compiler can statically check that data constraint). Generally, however, such unconditional effectiveness does not hold true. After all, if the B-graph for a data constraint φ in $|\boldsymbol{a}|$ has has no $\bigstar$-arborescence, commandification has no strict precedence relation to run Algorithm 2 with. In that case, $\mathsf{comm}(\varphi, X) = \varphi$, and consequently, commandification does not have its intended effect. Fortunately, commandification does satisfy a weaker—but still useful—form of effectiveness. To formulate this as a theorem, I first define a relation that holds true of *arborescent* constraint automata. I consider a constraint automaton arborescent if the B-graph for each of its data constraints has a $\bigstar$-arborescence.

Definition 59 (arborescentness)**.** $\clubsuit \subseteq \mathbb{A}\mathrm{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\left[\phi \in \mathsf{Dc}(\boldsymbol{a}) \ \textbf{implies} \ \lhd_{\phi}^{X} \neq \emptyset\right] \ \textbf{for all} \ \phi}{\clubsuit\, \boldsymbol{a}} \tag{7.35}$$

Note that by the previous definition, implicitly, an arborescent constraint automaton has only normal data constraints (otherwise, $\lhd_{\phi}^{X}$ does not exist for at least one $\phi \in \mathsf{Dc}(\boldsymbol{a})$ in Rule 7.35). The following theorem states the effectiveness of commandification, conditional on arborescentness: after commandifying an arborescent—hence normalized—constraint automaton $\boldsymbol{a}$, every data constraint in this automaton encodes a data command as a data relation (as in Definition 57). Let R range over the set of data relations defined in Definition 57 of comm.

Theorem 21. $\clubsuit\, \boldsymbol{a}$ **implies** $\mathsf{Dc}((\!|\boldsymbol{a}|\!)) \subseteq \{R(x_1, \ldots, x_k) \mid \mathbf{true}\}$

A FOCAML compiler can check constraint automata for arborescentness before applying commandification. To reduce the number of such checks, however, I also present a conjecture about preservation of arborescentness by operations on constraint automata. Before stating this conjecture, I first show

that multiplication generally does not preserve arborescentness. For instance, both $\mathsf{BinOp}\langle\mathtt{add}\rangle(\mathsf{A},\mathsf{B};\mathsf{C})$ and $\mathsf{Sync}(\mathsf{C};\mathsf{B})$ individually satisfy arborescentness, but their product does not. To see this, observe that the single transition in this product has $\mathtt{add}(\mathsf{A},\mathsf{B}) = \mathsf{C} \wedge \mathsf{C} = \mathsf{B}$ as its data constraint. One can easily verify that the B-graph for this data constraint indeed contains no arborescence, as one may have already expected: if a commandification of this data constraint *would* exist (which satisfaction of arborescentness would imply), that data command effectively assigns $\mathtt{add}(\mathsf{A},\mathsf{B})$ to B itself. Commandification, however, does not support such self-dependencies. This example instantiates a well-known problem in the theory and practice of synchronous systems: *causality loops*, "where the input is not known until the output is known, and the output can't be known until the input is known" [LNZ14]. In the previous example, Sync closes a causality loop between ports C and D. My conjecture about preservation of arborescentness by operations on constraint automata therefore states that multiplication preserves arborescentness only in the absence of causality loops.

Conjecture 2.

- $\big[\clubsuit\, \boldsymbol{a}_1, \boldsymbol{a}_2$ **and** $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2$ *has no causality loops*$\big]$ **implies** $\clubsuit\, \boldsymbol{a}_1 \otimes \boldsymbol{a}_2$
- $\big[\clubsuit\, \boldsymbol{a}$ **and** $p \notin \mathsf{Input}(\boldsymbol{a})\big]$ **implies** $\clubsuit\, \boldsymbol{a} \ominus p$
- $\big[\clubsuit\, \boldsymbol{a}$ **and** $p \notin \mathsf{Input}(\boldsymbol{a})\big]$ **implies** $\clubsuit\, \boldsymbol{a} \oplus p$

For now, I leave these preservation properties as a conjecture, because its truth or falsehood does not matter much in practice: although its (dis)proof would yield more insight in the theory of commandification, practical consequences remain insignificant. After all, this conjecture helps only in *predicting* when commandification may have its intended effect; it does not affect commandification's correctness whatsoever. Still, if this conjecture indeed holds true as I strongly suspect, a FOCAML compiler can accurately predict whether a product of primitives satisfies arborescentness, based on the arborescentness of these primitives, without again having to check this product for arborescentness (assuming that this product has no causality loops).

Before I did the work presented in this chapter, Clarke et al. already worked on purely constraint-based implementations of protocols [CPLA11]. Essentially, Clarke et al. specify not only the transition labels of an automaton as boolean constraints but also its state space and transition relation. In recent work, Proença and Clarke developed a variant of compile-time *predicate abstraction* to improve performance [PC13a]. They also used this technique to allow a form of interaction between a constraint solver and its environment during constraint solving [PC13b]. The work of Proença and Clarke resembles my work in the sense that we all try to "simplify" constraints at compile-time. I see also differences, though: (i) commandification fully avoids constraint solving and (ii) I adopted a richer language of data constraints in this thesis. For instance, Proença and Clarke have only unary functions in their language, which

would have cleared my need for B-graphs.

7.2 Practice

(I have not yet submitted the material in this section for publication.)

Compiler

I extended Lykos with the ability to apply commandification as in Figure 7.1, controllable through flag `COMMANDIFY`. When raised, Lykos first checks whether all primitives in the core set satisfy arborescentness. If so, by Conjecture 2, Lykos does not need to check products for arborescentness during the compilation process (as long as it encounters no causality loops). Note that even if Conjecture 2 turns out not to hold true, Lykos does not generate faulty code: in the worst case, Lykos unsuccessfully tries to commandify a constraint automaton, but once such commandification fails, Lykos simply defaults to the original data constraint. If this happens, compilation just takes a little longer. In contrast to Lykos's internals, the run-time library requires no modifications. Generated code looks a bit different if commandification succeeds, though, because in that case, Lykos has injected little pieces of sequential Java code—syntactically very similar to data commands—instead of calls to a constraint solver.

Experiments I: Protocols

I repeated the same experiments as in Chapter 6 (and Chapters 4 and 5), generating code for members of families SyncK, FifoK, Merger, Router, LateAsyncMerger, EarlyAsyncMerger, OddFibonacci, and Chess with the `COMMANDIFY`-flag raised, but otherwise under the same conditions as in Chapter 6. Figure 7.10 shows the per-family experimental results, averaged over five runs. The solid lines represent the actual measurements; the dotted lines represent inverse-proportional growth with respect to $k = 1$. The green lines represent the new results; the yellow lines represent the results from Chapter 6.

Figure 7.11 shows per-family speedup charts corresponding to the measurements in Figure 7.10; the dotted lines represent equal performance. For all constraint automata with which I experimented, except members of Merger and Router, commandification indeed improves performance, to a lesser or to a greater extent. The code generated for Mergers and Routers has, overall, similar performance with and without commandification (because these constraint automata have very simple data constraints, which cost relatively few computational resources compared to their nondeterministic choice among increasingly many options). Because of the scale on the y-axis, the speedup for members of OddFibonacci seems almost nonexistent in Figure 7.10, but in fact, Figure 7.11 shows that speedup ranges from 28% (for $k = 64$) up to 275%

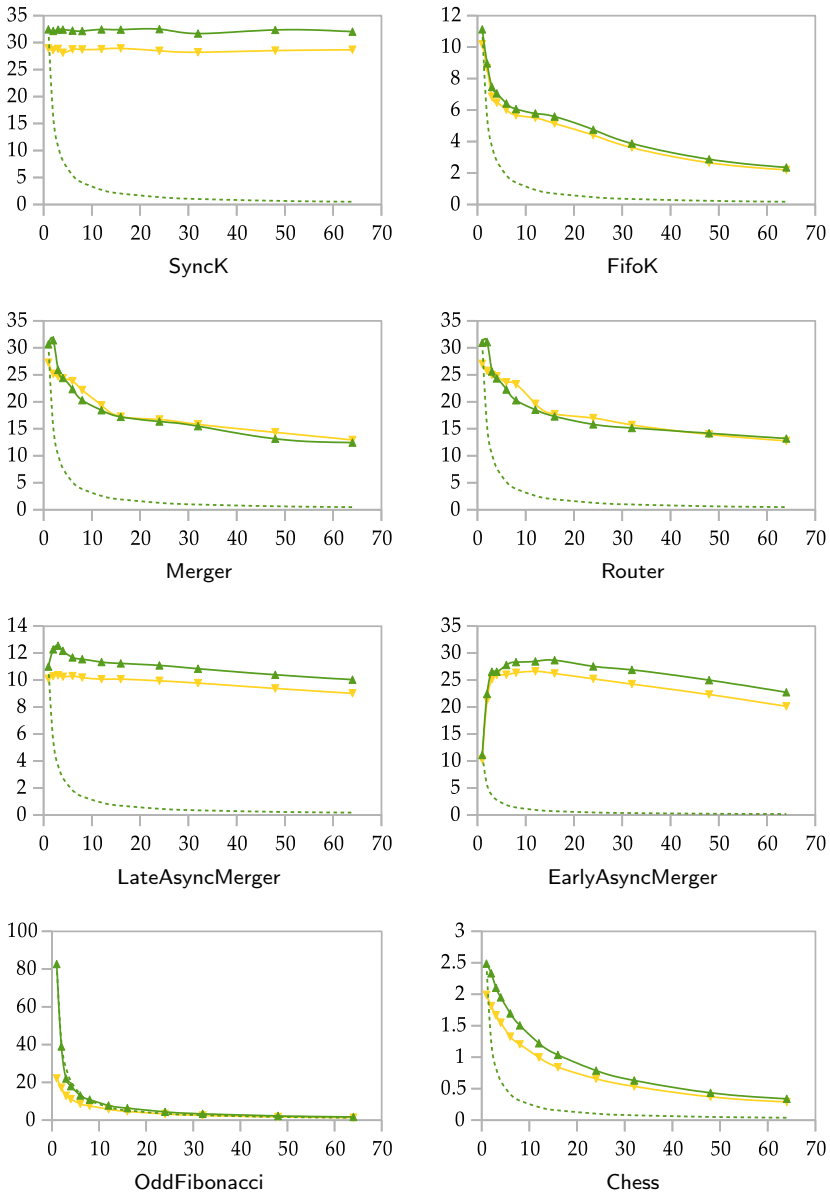


Figure 7.10: Performance (in number of completed rounds per four minutes) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

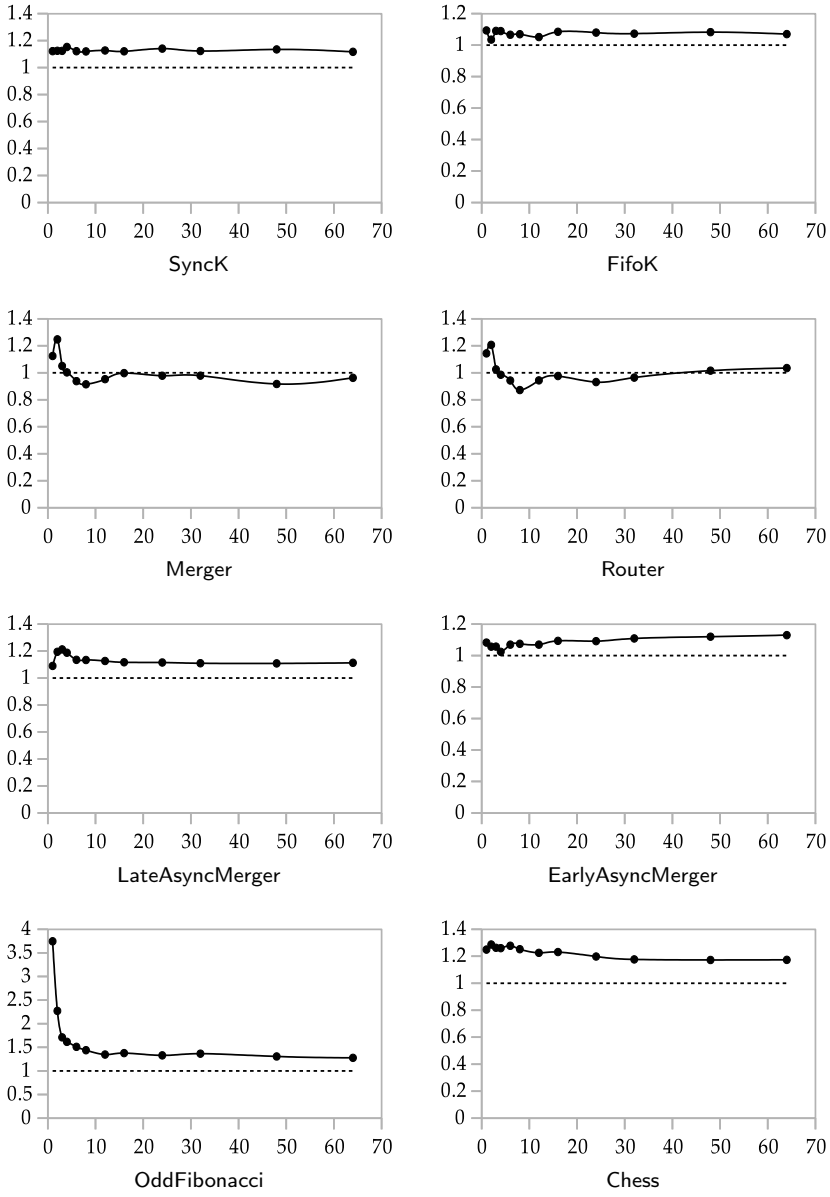


Figure 7.11: Speedup (relative to compiler-generated code in Chapter 6) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

(for $k = 1$). This makes commandification actually most effective for OddFibonacci (because members of OddFibonacci require relatively complex data processing).

Experiments II: Programs

I repeated the same experiments as in Chapter 6 (and Chapters 4 and 5), generating code for the NPB benchmarks with the `COMMANDIFY`-flag raised, but otherwise under the same conditions as in Chapter 6.

Figures 7.12–7.19 show performance charts for the FOCAML-to-Java-compiled versions of the NPB kernel benchmarks (averaged over five runs), speedup charts (with respect to their Java versions by Frumkin et al.), and charts about cache misses. The dotted green/yellow lines represent the MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks with and without commandification (i.e., the green lines represent the new results, while the yellow lines represent the results in Chapter 6); the solid green/yellow lines represent the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions; the dotted black lines represent the Java versions by Frumkin et al.

I make the following main observations about these experimental results:

- Overall, the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks outperform their MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions (solid lines versus dotted lines), as in Chapters 5 and 6. Furthermore, the FOCAML-to-Java-compiled versions with commandification perform at least as well as the FOCAML-to-Java-compiled versions without commandification (green lines versus yellow lines), and often better by a small margin.

- Although the Java versions of the NPB kernel benchmarks by Frumkin et al. still slightly outperform many of their FOCAML-to-Java-compiled versions, the margin has further decreased (compared to the results in Chapter 6), albeit by only a little: after syntactic subtraction, commandification does not make as big an impact.

- The same point about cache misses made in Chapter 5 applies here too: numbers of cache misses seem a fair indicator of performance.

- The same point about increasing problem sizes made in Chapter 5 applies here too: as the problem size increases, the speedup generally improves.

- As in Chapters 5 and 6, differences in numbers of cache misses explain why the FOCAML-to-Java-compiled versions of NPB-IS without commandification outperform their supposedly improved FOCAML-to-Java-compiled versions with commandification for $k = 64$ in class W, class A, and class B.

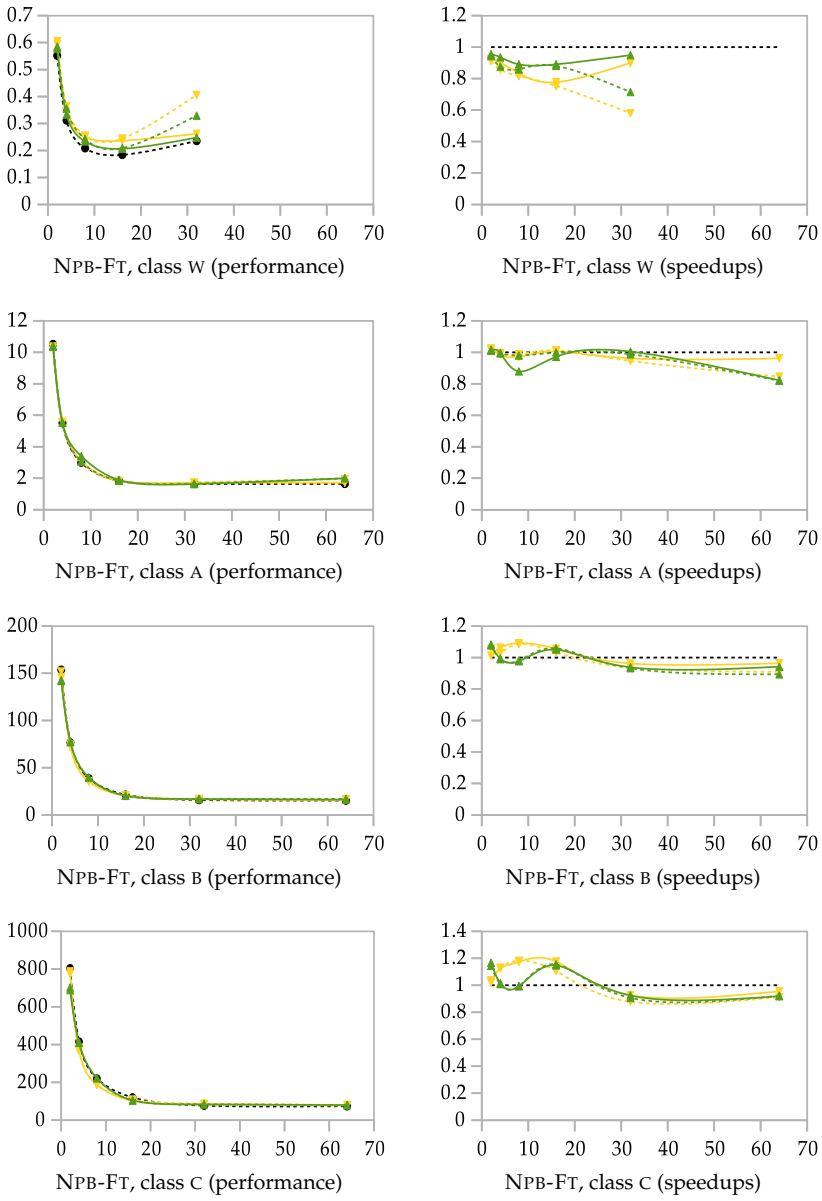


Figure 7.12: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

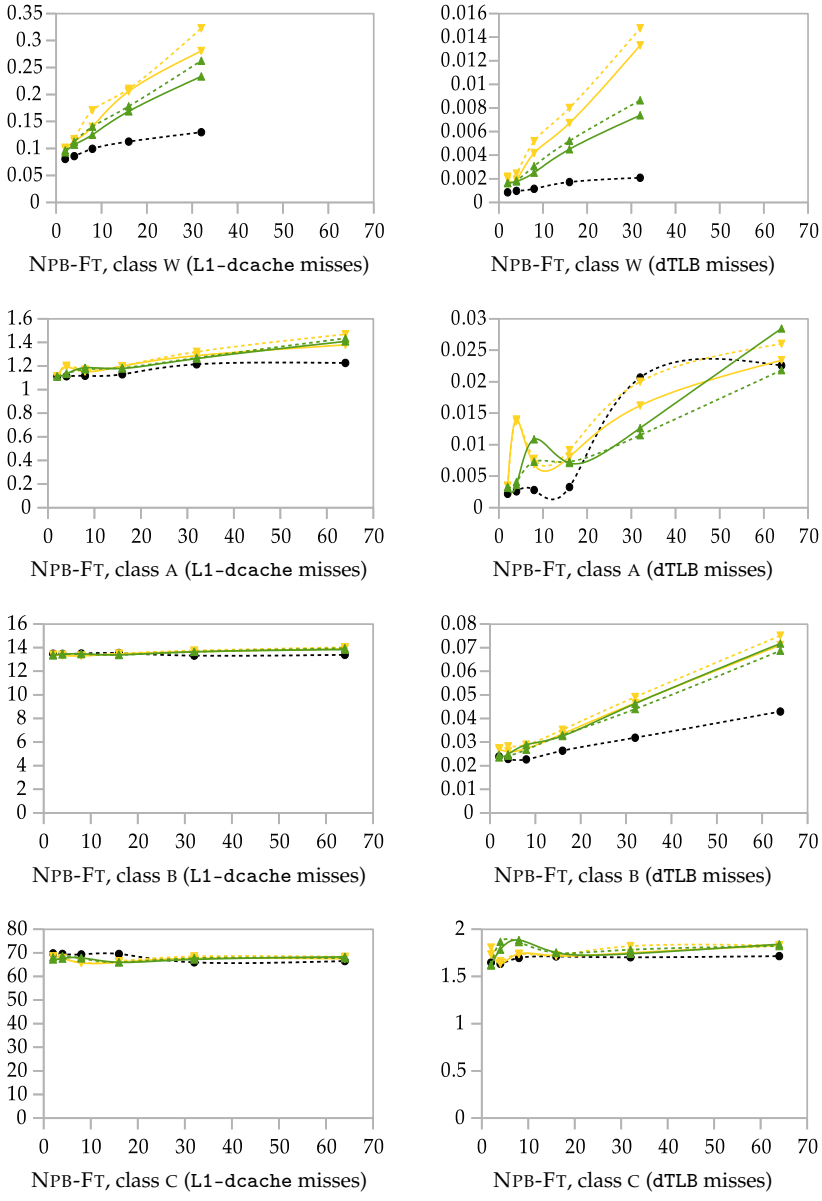


Figure 7.13: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

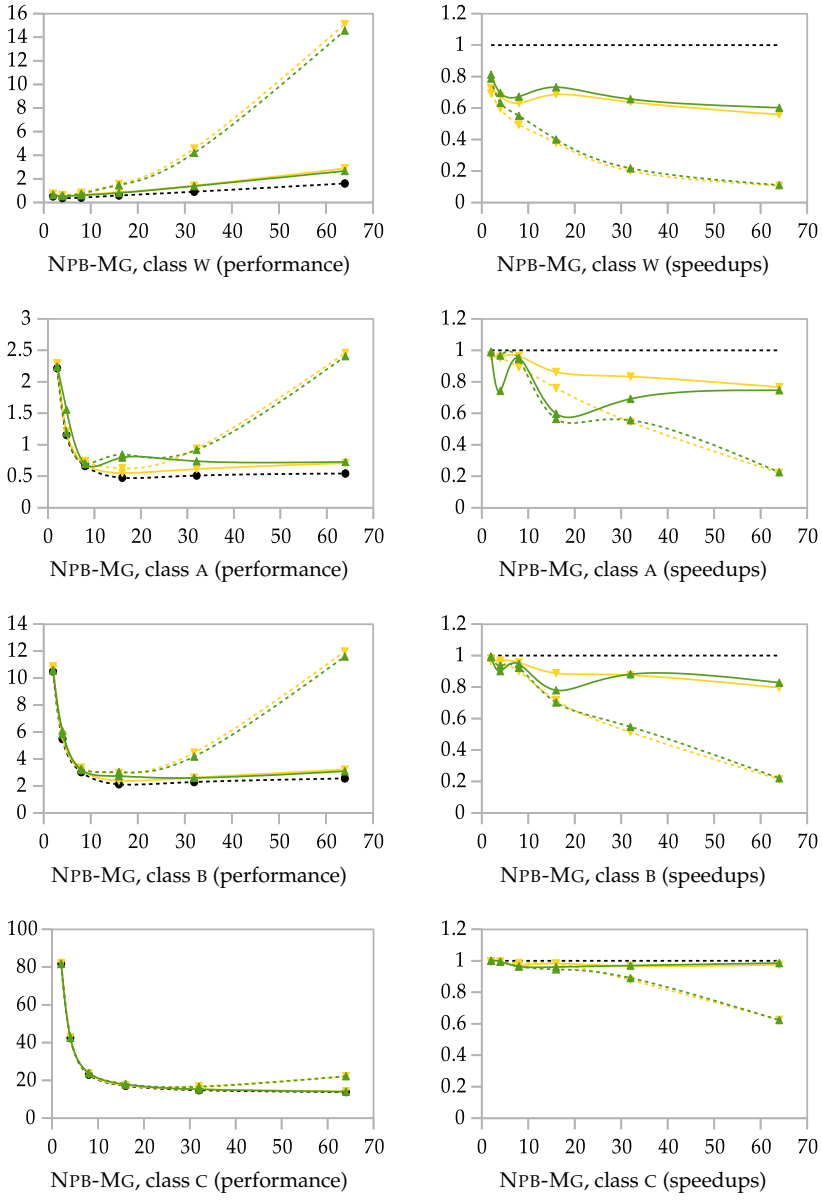


Figure 7.14: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

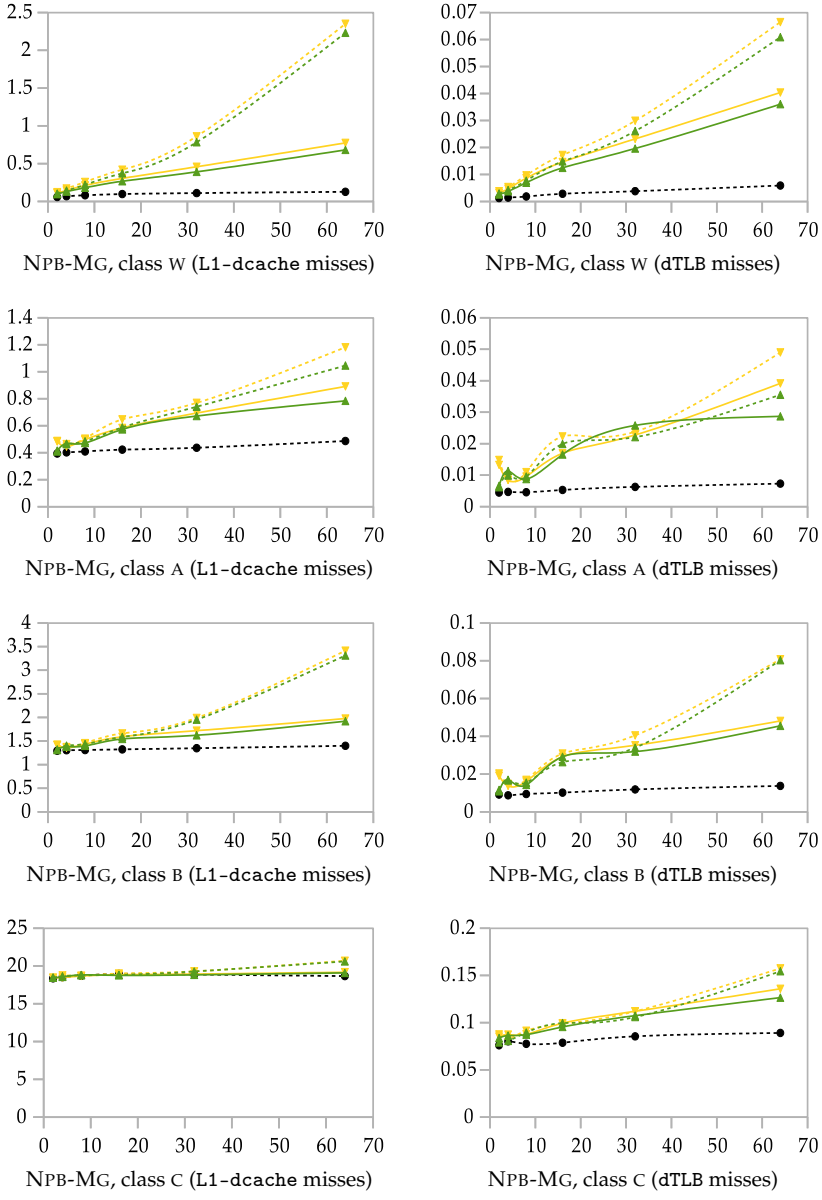


Figure 7.15: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

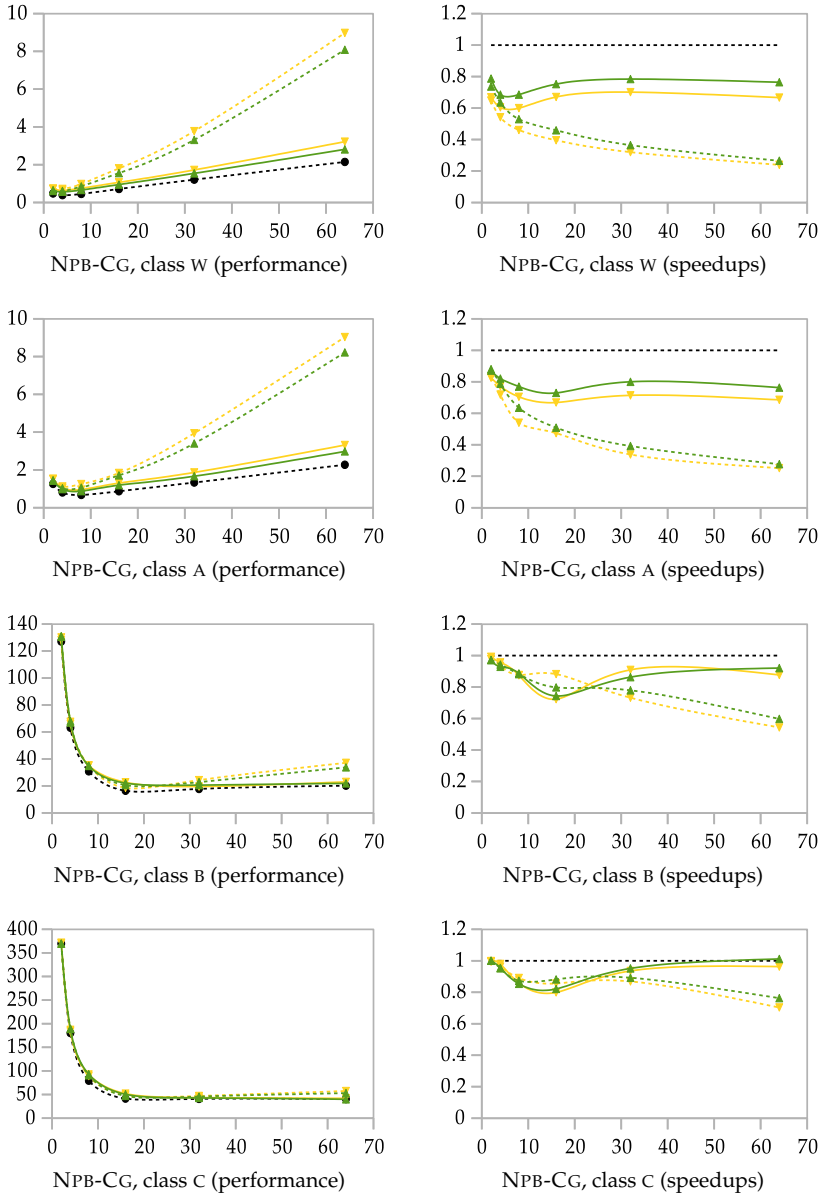


Figure 7.16: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

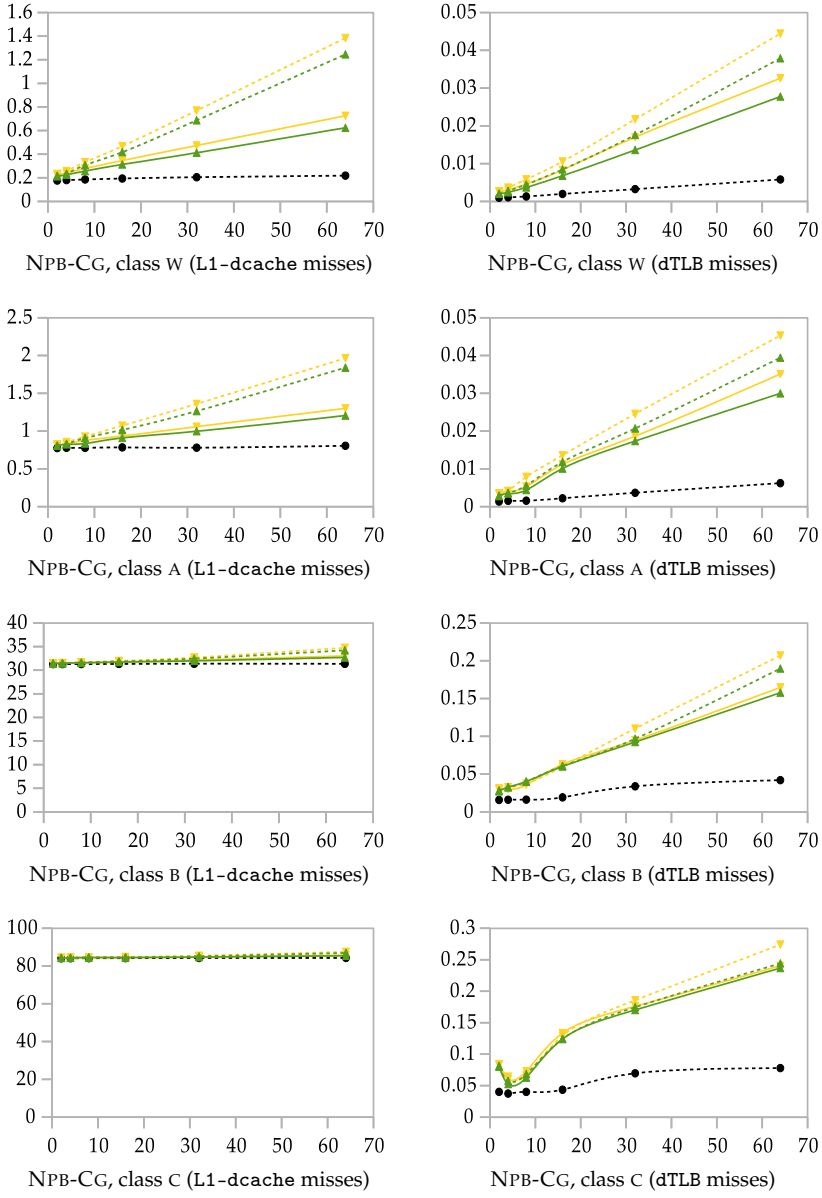


Figure 7.17: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

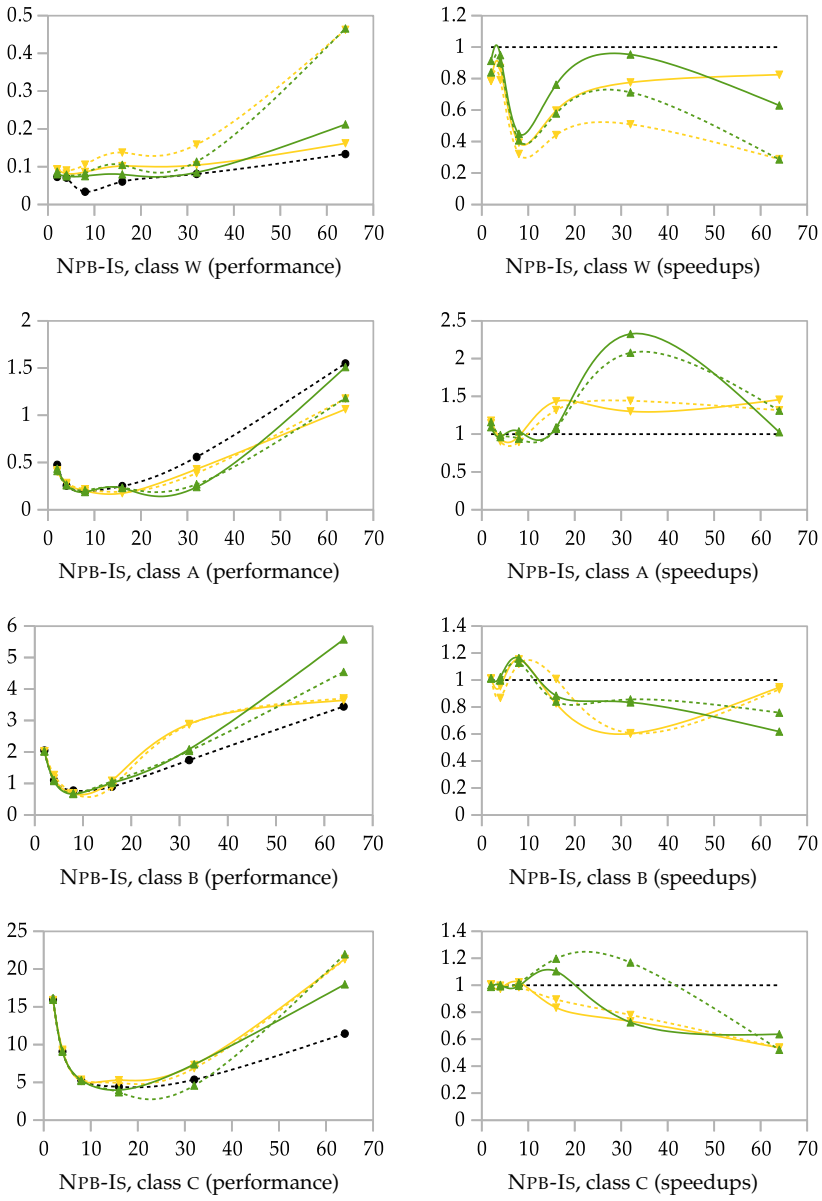


Figure 7.18: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

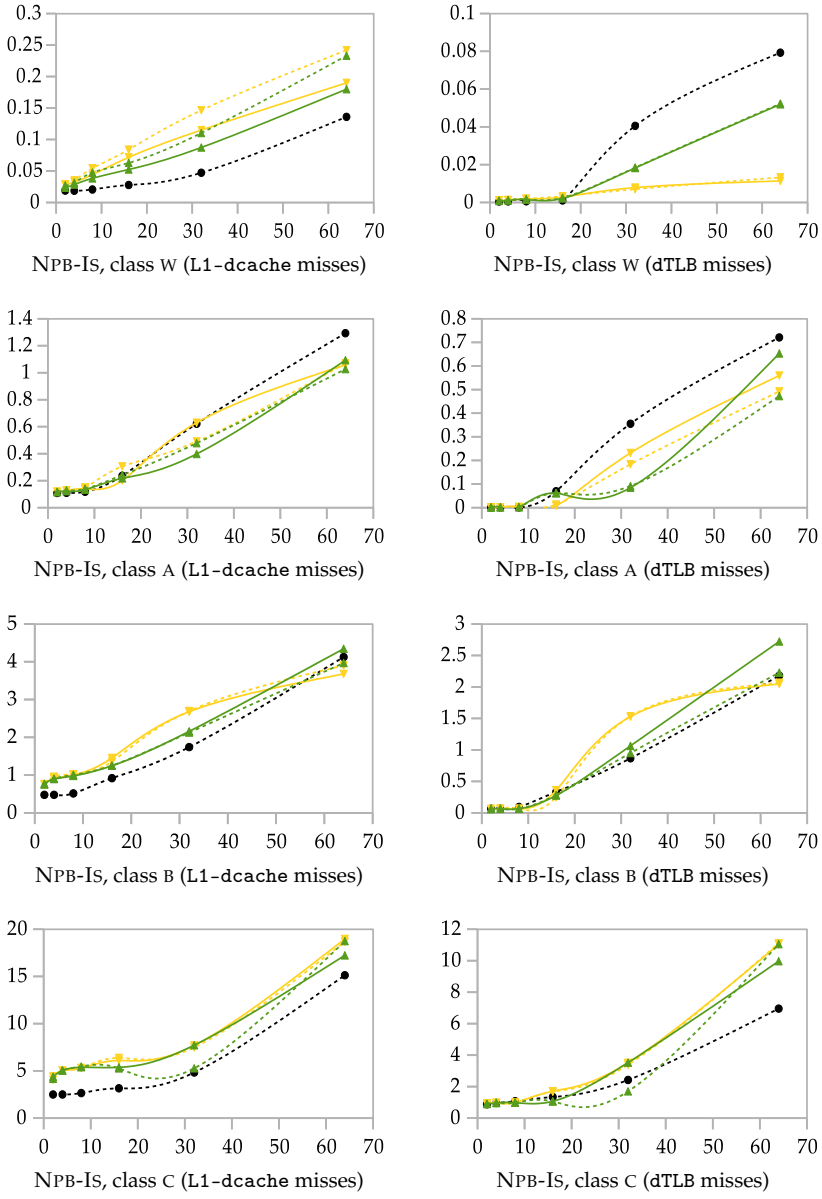


Figure 7.19: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

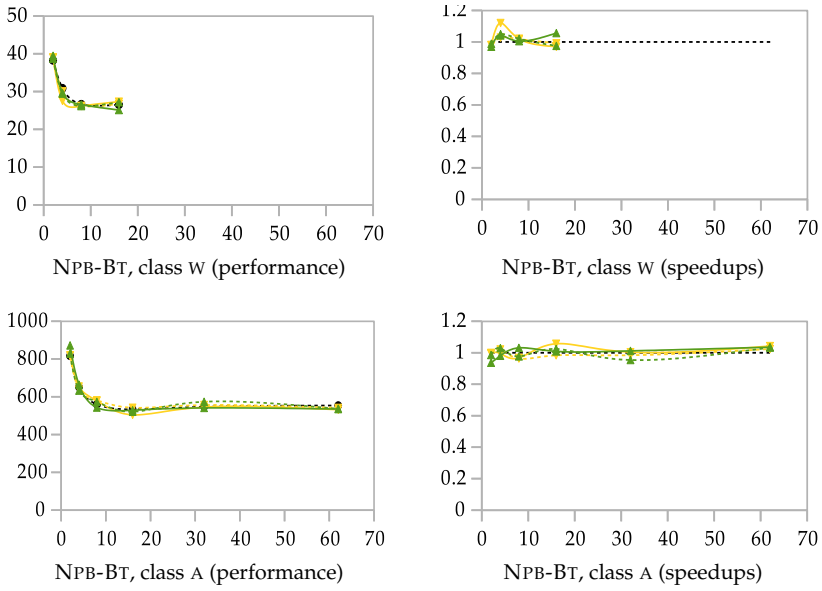


Figure 7.20: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

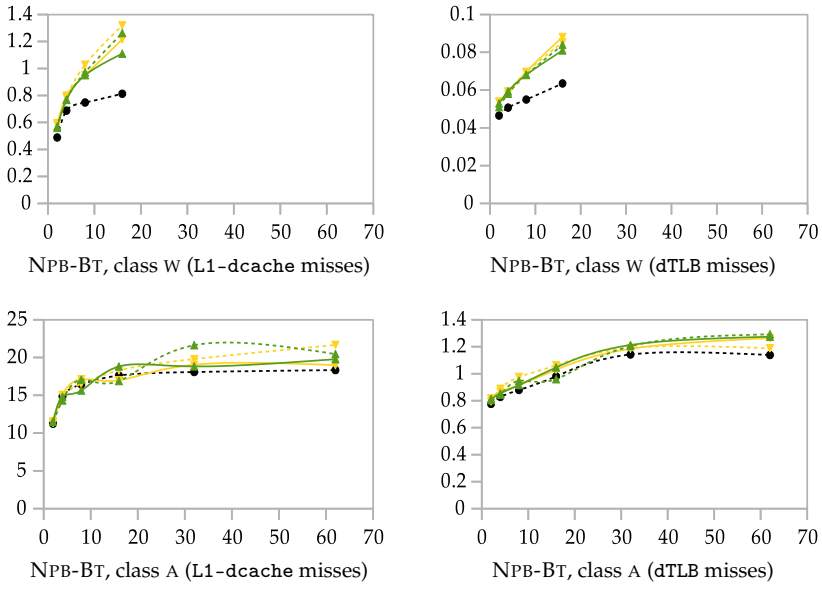


Figure 7.21: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

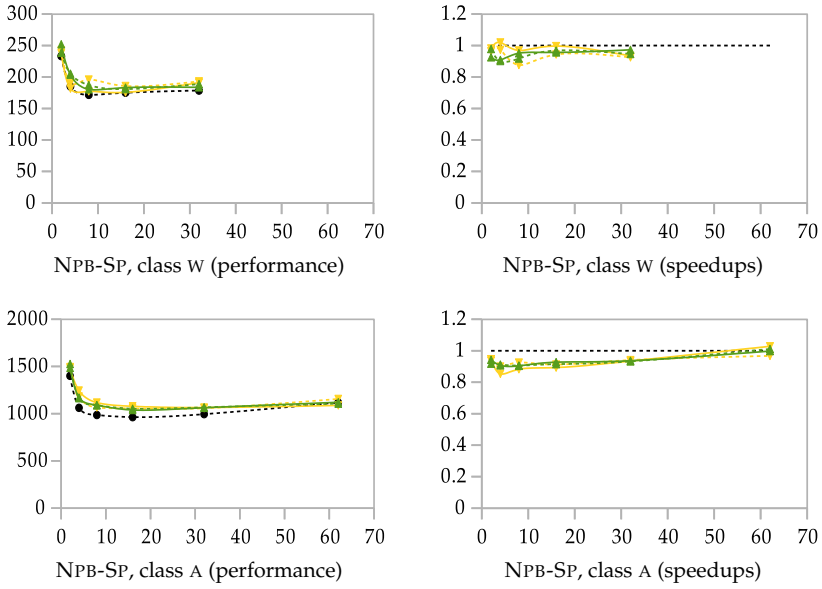


Figure 7.22: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

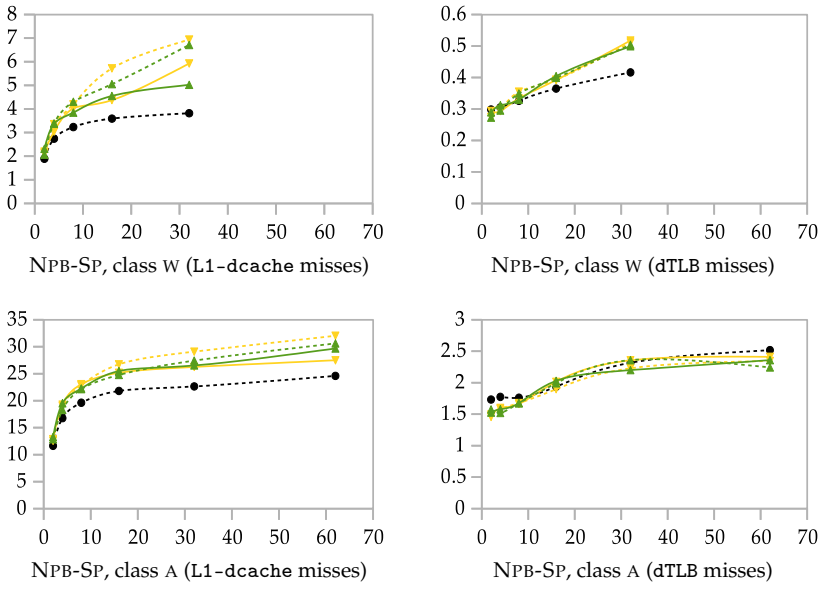


Figure 7.23: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

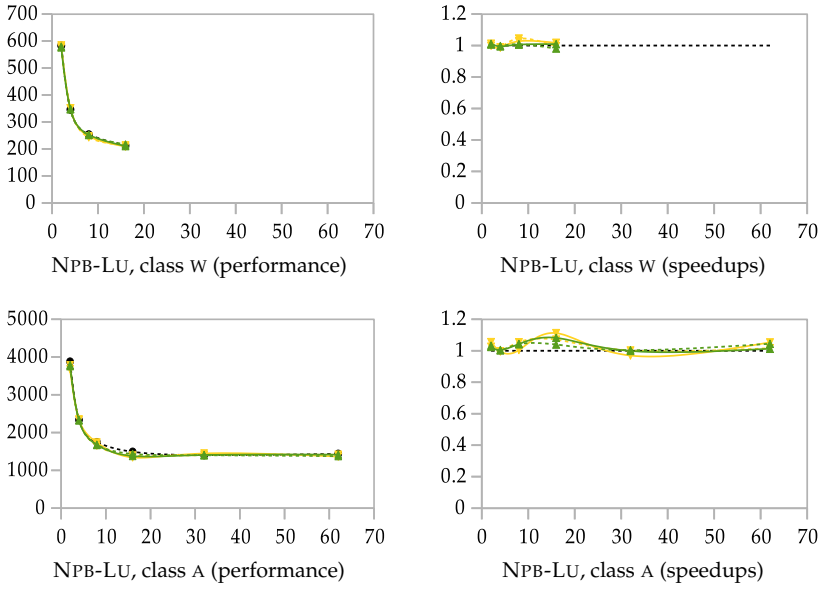


Figure 7.24: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

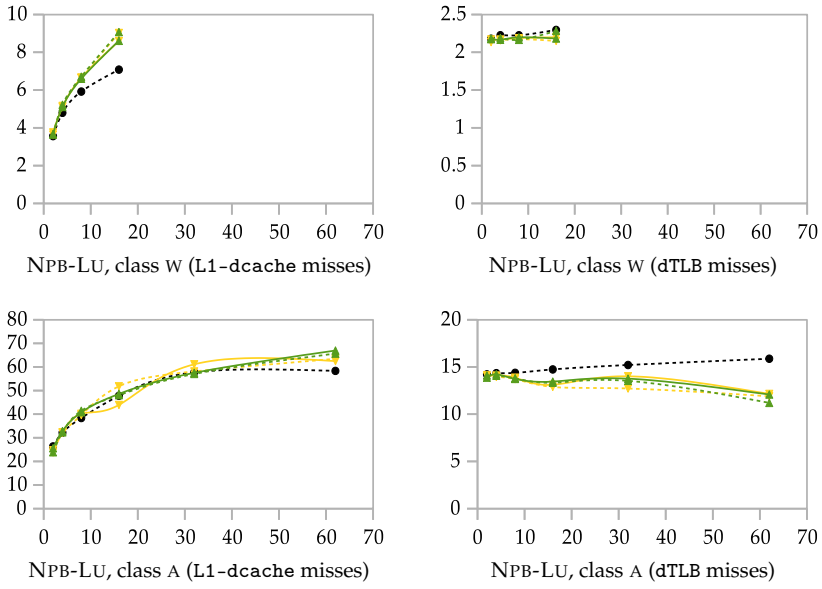


Figure 7.25: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

Figures 7.20–7.25 show performance charts for the FOCAML-to-Java-compiled versions of the NPB application benchmarks (averaged over five runs), speedup charts (with respect to their Java versions by Frumkin et al.), and charts about cache misses. The lines have the same meaning as in the figures with experimental results for the NPB kernel benchmarks. Recall from Figure 5.12 that NPB-BT and NPB-LU do not support more than 22 and 31 slaves, for which reason I have no measurements beyond $k = 16$ in class W for those benchmarks. In the same figure, note that NPB-BT, NPB-SP, and NPB-LU support at most 62 workers in class A. For that reason, as in Chapter 6, I compiled the FOCAML versions of those benchmarks for $k = 62$ instead of $k = 64$. Essentially, the same observations apply here as for the previous experimental results of the NPB kernel benchmarks.

Chapter 8

Improved Compilation IV: Queue-Inference

In Chapter 6, I argued that the performance of compiler-generated code for Mergers, Routers, LateAsyncMergers, and EarlyAsyncMergers should stay (close to) constant in the number of producers/consumers. The experimental results in Chapter 7, however, show that code generated by Lykos does not achieve such scalability, not even with commandification: compiler-generated code for LateAsyncMergers and EarlyAsyncMergers benefits from commandification by 10–20%, but without improving scalability, while compiler-generated code for Mergers and Routers does not seem to benefit from commandification at all.

In this chapter, I present a technique, called *queue-inference*, that improves the scalability of compiler-generated code for protocols with nondeterministic choices, such as Merger, Router, LateAsyncMerger, and EarlyAsyncMerger. In Section 8.1, to develop an intuition for what queue-inference involves, I first explain how to manually apply this optimization technique. Subsequently, I formalize queue-inference in terms of constraint automata, thereby making this technique amenable to automation. In Section 8.2, I present an improved version of Lykos using queue-inference, including new experimental results on performance.

Although the improvement presented in this chapter eventually results in improved compiler-generated code, as in Chapters 5, 6 and 7, I define this improvement at the higher level of constraint automata instead of at the lower level of GPL code. Not only does this facilitate more elegant formal reasoning about correctness (compared to reasoning directly about GPL code), but it also eases the automatic application of this improvement by a FOCAML compiler. Moreover, it makes this improvement independent of GPLs—Java in this thesis—so that the same optimization automatically applies to, for instance, generated C code.

8.1 Theory

*(With Arbab and Halle, I previously published fragments of material in this section in a conference paper [**J**HA14a].)*

Manual Optimization

To understand why the compiler-generated code for Mergers, Routers, LateAsyncMergers, and EarlyAsyncMergers in Chapter 7 scales suboptimally, recall the following run-time behavior of worker and protocol units from Chapter 4: whenever a worker unit performs an I/O operation on the data structure for a port, it informs the protocol unit that shares access to this data structure about this event, after which this protocol unit starts a new round of event-handling. As shown in the simplified event-handler in Figure 4.6, this protocol unit subsequently loops over all transitions out of the current state in search of an enabled one and, once found, fires this enabled transition. For k outgoing transitions, then, this loop requires $\mathcal{O}(k)$ time; clearly, as the number of transitions increases, the average time taken by a protocol unit to complete one round of event-handling also increases. Because the number of transitions in Mergers, Routers, LateAsyncMergers, and EarlyAsyncMergers increases linearly in the number of producers/consumers, exactly the linear complexity of event-handling causes compiler-generated code for those constraint automata to scale suboptimally. This analysis, although formulated here for the Centralized Approach, similarly applies to the Hybrid Approach.

To find a solution for this scalability problem, or at the very least some inspiration, suppose that I provide EarlyAsyncMerger—a representative instance of this problem—as a specification to software engineers and ask them for a manual implementation. I actually did this little exercise with Sean Halle, then a colleague at CWI with many years of experience in parallel programming. Sean made two implementations. In his first implementation, every producer has its own variable for storing data-to-send. To receive, then, the consumer needs to iterate over all these variables in search of a nonempty one (cf., looping over all outgoing transitions in a constraint automaton by event-handlers). Of course, Sean also used locks (with condition variables) to synchronize the producers/consumer and avoid race conditions, but I skip those here. In Sean's first implementation, thus, once the consumer receives a datum, it actually knows which specific producer sent that datum (namely, the producer corresponding to the nonempty variable in which the consumer found that datum). This, however, overimplements my intention: the consumer does not really care about which *specific* producer it receives from so long as it receives from *some* producer. In other words, the producers may remain *indistinguishable* to the consumer. Sean's second implementation exploits this indistinguishability. Instead of using per-producer variables, in this second implementation, the producers offer their data-to-send into a *queue*. To receive, the consumer can simply poll a datum from the queue in $\mathcal{O}(1)$ time (ignoring, for simplicity, the overhead of synchronizing concurrent queue accesses). In this second im-

```
1  public interface Queue {
2    public boolean isEmpty();
3    public void offer(Port port);
4    public Port peek();
5    public Port poll();
6  }

7  public class QueueBasedOutputPortImpl extends OutputPortImpl {
8    public volatile Queue queue;
9
10   @Override
11   public void put(Object datum) throws InterruptedException {
12     buffer = datum;
13     status = IO.PENDING;
14     handler.register();
15     queue.offer(this);
16     resume();
17 } }

18 public class QueueBasedInputPortImpl extends InputPortImpl {
19   public volatile Queue queue;
20
21   @Override
22   public Object get() throws InterruptedException {
23     buffer = null;
24     status = IO.PENDING;
25     handler.register();
26     queue.offer(this);
27     return resume();
28 } }
```

Figure 8.1: Java run-time library extended with queues

plementation, thus, the consumer never knows from which specific producer it receives.

Although not directly a solution, Sean's second implementation formed a key inspiration for the optimization technique presented in this chapter: to improve scalability, compiler-generated code should leverage indistinguishability among workers by using queues. To clarify what exactly this means in the context of compiler-generated code for constraint automata, in the rest of this subsection, I explain—by example—how to perform this optimization technique by manually introducing queues in previous unoptimized compiler-generated code. In subsequent subsections, then, I formalize and automate this transformation at the higher level of constraint automata.

I take the compiler-generated code in Chapter 4 as my starting point. Although Lykos generated that code under the Centralized Approach, code generated under the Hybrid Approach requires similar modifications. First, Figure 8.1 shows my modifications to the run-time library. (This figure constitutes one of the rare exceptions in this thesis, where I qualify ports as "input" or "output" from the perspective of workers.) Implementations of interface `Queue` comprise queue data structures for `Ports`. Whenever a thread `peeks`

```
1  public class QueueBasedProtocol extends Protocol {
2    public Protocol(Port A, Port B, Port C) {
3      super(A, B, C);
4    }
5
6    @Override
7    public void initialize() {
8      ((QueueBasedOutputPortImpl) A).queue = automaton7.queue;
9      ((QueueBasedOutputPortImpl) B).queue = automaton7.queue;
10     super.initialize();
11 } }
```

```
1  class QueueBasedAutomaton7 extends Automaton7 {
2    final Queue queue = new Queue();
3  }
```

Figure 8.2: Classes `QueueBasedProtocol` and `QueueBasedAutomaton7`, manually derived from the automatically generated classes `Protocol` and `Automaton7` in Figures 4.15 and 4.16

such a data structure, it reads its first element without removing it; whenever a thread `poll`s, it not only reads but also removes. Classes `QueueBasedOutputPortImpl` and `QueueBasedInputPortImpl` extend classes `OutputPortImpl` and `InputPortImpl` in Figure 4.13 with `Queue` fields (set elsewhere, discussed shortly) and, notably, with invocations of method `offer` on lines 15 and 26. Thus, whenever a thread performs an I/O operation on a `Port`, after setting all the fields of that `Port`, it `offer`s this `Port` into the `queue`.

Figure 8.2 shows my manual modifications to the automatically generated classes `Protocol` and `Automaton7` in Figures 4.15 and 4.16. In method `initialize` of `QueueBasedProtocol`, the current thread sets the `queue`s in `Port`s `A` and `B`, in addition to everything it already had to do. The value assigned to `A.queue` and `B.queue` comes from an instance of `QueueBasedAutomaton7`, which differs from instances of `Automaton7` only in the added `Queue` field.

Finally, I also manually modified the automatically generated classes `Automaton7Transition1` and `Automaton7Transition2` in Figures 4.18 and 4.19. Figure 8.3 shows these modifications. Class `QueueBasedAutomaton7Transition1` has a new `Queue` field, `initialized` to the `queue` of the `Automaton7` in the `Protocol`. In method `checkSynchronizationConstraint`, instead of using the `Context` of the `Automaton7` as in method `checkSynchronizationConstraint` of the original `Automaton7Transition1`, the current thread checks if the `queue` contains a `Port`. After all, by my previous modifications to the runtime library, if a thread has performed an I/O operation on a `Port`, it must have offered that `Port` into the `queue`. In method `fire`, the current thread actually does the same as in method `fire` of the original `Automaton7Transition1`, except that it first needs to `poll` the `queue` to get the actual `Port` to operate on. To highlight their differences, I grayed out the similar parts in class `QueueBasedAutomaton7Transition2` with respect to class `QueueBasedAutomaton7Transition1`. These classes differ only in their name, which has the following

```
class QueueBasedAutomaton7Transition1 extends Automaton7Transition1 {
  Queue queue;

  public initialize(Protocol protocol) {
    this.queue = protocol.automaton7.queue;
    super.initialize(protocol);
  }

  @Override
  protected boolean checkSynchronizationConstraint() {
    return true && !queue.isEmpty();
  }

  @Override
  protected boolean fire() {
    boolean canFire = checkSynchronizationConstraint() && checkDataConstraint();
    if (canFire) {
      Port port = queue.poll();
      port.status = IO.COMPLETED;
      port.semaphore.release();
      target.reach();
    }
    return canFire;
} }

class QueueBasedAutomaton7Transition2 extends Automaton7Transition2 {
  Queue queue;

  public initialize(Protocol protocol) {
    this.queue = protocol.automaton7.queue;
    super.initialize(protocol);
  }

  @Override
  protected boolean checkSynchronizationConstraint() {
    return true && !queue.isEmpty();
  }

  @Override
  protected boolean fire() {
    boolean canFire = checkSynchronizationConstraint() && checkDataConstraint();
    if (canFire) {
      Port port = queue.poll();
      port.status = IO.COMPLETED;
      port.semaphore.release();
      target.reach();
    }
    return canFire;
} }
```

Figure 8.3: Classes `QueueBasedAutomaton7Transition1` and `QueueBasedAutomaton7Transition2`, manually derived from the automatically generated classes `Automaton7Transition1` and `Automaton7Transition2` in Figures 4.18 and 4.19

important consequence.

Reconsider class `HandlerForABC` in Figure 4.22 in the context of my previous modifications. Every instance of this `Handler` represents a comprehensive event-handler for `QueueBasedAutomaton7`, which any thread can call at any time in an attempt to `fire` any `Transition` (as described in more detail in Chapter 4). In method `call` of `HandlerForABC`, the current thread loops over an array of `Transitions` until it has successfully `fired` one. In the context of my previous modifications, this array consists of a `QueueBasedAutomaton7Transition1` and a `QueueBasedAutomaton7Transition2`. However, as highlighted in Figure 4.22, these classes consist of exactly the same code. Therefore, storing an instance of each of those classes in the array effectively amounts to letting the current thread try to `fire` the same `Transition` twice. To avoid this, I may modify `HandlerForABC` by removing the loop and letting the current thread invoke either method `fire` of `QueueBasedAutomaton7Transition1` or method `fire` of `QueueBasedAutomaton7Transition2`—but never both. By doing so, the current thread effectively tries to `fire` one of two `Transitions` at the same time.

With these modifications, essentially, I exploit the indistinguishability of producers to the consumer by making the ports on which those producers perform their I/O operations indistinguishable: without extra information about which producer has access to which specific port, afterward reconstructing which producer offered a port into a queue becomes impossible, notably upon polling that port from the queue—the ports have become indistinguishable. From an automata-theoretic perspective, my modifications to previous compiler-generated code essentially correspond to the notion of *combining* multiple transitions into a single transition (which, at run-time, requires only one check for enabledness in every round of event-handling by using queues). In the next subsections, I present a formalization of this automata-theoretic perspective, including a formalization of port indistinguishability.

Multiconstraint Automata

In this subsection and the next, I call constraint automata as defined in Chapter 2, Definition 19, *uniconstraint automata*. This new piece of terminology allows me to clearly distinguish uniconstraint automata from *multiconstraint automata*, which generalize uniconstraint automata and support combining multiple transitions into single transitions in a behavior-preserving way (which uniconstraint automata do not support). Multiconstraint automata differ from uniconstraint automata in only one aspect: while uniconstraint automata have *synchronization uniconstraints*—sets of ports as in Definition 19—in their transition labels, multiconstraint automata have *synchronization multiconstraints*. Henceforth, I call transitions in uniconstraint automata *unitransitions* and transitions in multiconstraint automata *multitransitions*. See the last paragraph of this subsection for related work.

Definition 60 (multiconstraint automata)**.** *A multiconstraint automaton is a tuple:*

$$(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0))$$

where:

- $Q \subseteq \mathbb{Q}$ *(states)*
- $(P^{\text{all}}, P^{\text{in}}, P^{\text{out}}) \in 2^{\mathbb{P}} \times 2^{\mathbb{P}} \times 2^{\mathbb{P}}$ *such that:* *(ports)*

$$P^{\text{in}}, P^{\text{out}} \subseteq P^{\text{all}} \ \textbf{and} \ P^{\text{in}} \cap P^{\text{out}} = \emptyset$$

- $M \subseteq \mathbb{M}$ *(memory cells)*
- $\longrightarrow \subseteq Q \times 2^{2^{2^{P^{\text{all}}}}} \times \mathsf{Good}(P^{\text{all}} \cup {}^{\bullet}M \cup M^{\bullet}) \times Q$ *(multitransitions)*
- $(q^0, \mu^0) \in Q \times (M \to \mathbb{D})$ *(initial configuration)*

$\mathbb{A}\text{UTOM}^{+}$ *denotes the set of all multiconstraint automata, ranged over by* $\boldsymbol{e}$*.*

Every synchronization multiconstraint consists of $\big[$a set G of $\big[$sets E_i of $\big[$sets V_{ij} of ports$\big]\big]\big]$ and represents a nondeterministic choice among synchronization uniconstraints. As a first few examples,

$$G = \{\{\overbrace{\{\mathsf{A}\}}^{V_{11}}, \overbrace{\{\mathsf{D}\}}^{V_{12}}\}\}_{E_1} \quad \text{and} \quad G = \{\{\overbrace{\{\mathsf{A}\}}^{V_{11}}, \overbrace{\{\mathsf{C}\}}^{V_{12}}, \overbrace{\{\mathsf{E}\}}^{V_{13}}\}\}_{E_1}$$

represent the synchronization uniconstraints $\{\mathsf{A}, \mathsf{D}\}$ and $\{\mathsf{A}, \mathsf{C}, \mathsf{E}\}$ (i.e., no nondeterministic choice). As a second few examples,

$$G = \{\underbrace{\{\overbrace{\{\mathsf{A}\}}^{V_{11}}, \overbrace{\{\mathsf{D}\}}^{V_{12}}\}}_{E_1}, \underbrace{\{\overbrace{\{\mathsf{B}\}}^{V_{21}}, \overbrace{\{\mathsf{D}\}}^{V_{22}}\}}_{E_2}, \underbrace{\{\overbrace{\{\mathsf{C}\}}^{V_{31}}, \overbrace{\{\mathsf{D}\}}^{V_{32}}\}}_{E_3}\}$$

and

$$G = \{\underbrace{\{\overbrace{\{\mathsf{A}\}}^{V_{11}}, \overbrace{\{\mathsf{C}\}}^{V_{12}}, \overbrace{\{\mathsf{E}\}}^{V_{13}}\}}_{E_1}, \underbrace{\{\overbrace{\{\mathsf{A}\}}^{V_{21}}, \overbrace{\{\mathsf{D}\}}^{V_{22}}, \overbrace{\{\mathsf{E}\}}^{V_{23}}\}}_{E_2}, \underbrace{\{\overbrace{\{\mathsf{B}\}}^{V_{31}}, \overbrace{\{\mathsf{C}\}}^{V_{32}}, \overbrace{\{\mathsf{E}\}}^{V_{33}}\}}_{E_3}, \underbrace{\{\overbrace{\{\mathsf{B}\}}^{V_{41}}, \overbrace{\{\mathsf{D}\}}^{V_{42}}, \overbrace{\{\mathsf{E}\}}^{V_{43}}\}}_{E_4}\}$$

represent a nondeterministic choice among the three synchronization uniconstraints $\{\mathsf{A}, \mathsf{D}\}$, $\{\mathsf{B}, \mathsf{D}\}$, $\{\mathsf{C}, \mathsf{D}\}$ and a nondeterministic choice among the four synchronization uniconstraints $\{\mathsf{A}, \mathsf{C}, \mathsf{E}\}$, $\{\mathsf{A}, \mathsf{D}, \mathsf{E}\}$, $\{\mathsf{B}, \mathsf{C}, \mathsf{E}\}$, $\{\mathsf{B}, \mathsf{D}, \mathsf{E}\}$. Thus, in the previous examples—and also in every other synchronization multiconstraint whose every V_{ij} contains exactly one port—G represents a disjunction (with disjuncts represented by E_i) of conjunctions (with conjuncts represented

by V_{ij}). Finally, to explain the more difficult meaning of synchronization multiconstraints with nonsingleton V_{ij}'s, as a third few examples,

$$G = \{\underbrace{\{\overbrace{\{\mathsf{A},\mathsf{B},\mathsf{C}\}}^{V_{11}},\overbrace{\{\mathsf{D}\}}^{V_{12}}\}}_{E_1}\} \quad \text{and} \quad G = \{\underbrace{\{\overbrace{\{\mathsf{A},\mathsf{B}\}}^{V_{11}},\overbrace{\{\mathsf{C},\mathsf{D}\}}^{V_{12}},\overbrace{\{\mathsf{E}\}}^{V_{13}}\}}_{E_1}\}$$

represent *exactly* the same nondeterministic choices among synchronization uniconstraints as the previous two synchronization multiconstraints. Thus, while G represents a disjunction and while every $E_i \in G$ represents a conjunction, every $V_{ij} \in E_i$ represents a *uniqueness quantification*: to compute the nondeterministic choice among synchronization uniconstraints represented by G, for every E_i, collect all synchronization uniconstraints (i.e., sets of ports) constructed by selecting exactly one port from every V_{ij}.

Definition 61 (interpretation of synchronization multiconstraints)**.** $\|\cdot\| : 2^{2^{2^{\mathbb{P}}}} \cup 2^{2^{\mathbb{P}}} \to 2^{\mathbb{P}}$ *denotes the function defined by the following equation:*

$$\|E\| = \left\{ \{p_1,\ldots,p_n\} \,\middle|\, \begin{array}{l} E = \{V_1,\ldots,V_n\} \\ \textbf{and } p_1 \in V_1 \textbf{ and } \cdots \textbf{ and } p_n \in V_n \end{array} \right\}$$
$$\|G\| = \bigcup\{\|E\| \mid E \in G\}$$

Subsequently, I straightforwardly map every multiconstraint automaton to a uniconstraint automaton.

Definition 62 (interpretation of multiconstraint automata)**.** $\|\cdot\| : \mathbb{A}\text{UTOM}^+ \to \mathbb{A}\text{UTOM}$ *denotes the function defined by the following equation:*

$$\|(Q,(P^{\text{all}},P^{\text{in}},P^{\text{out}}),M,\longrightarrow,(q^0,\mu^0))\| = (Q,(P^{\text{all}},P^{\text{in}},P^{\text{out}}),M,\|\longrightarrow\|,(q^0,\mu^0))$$

where $\|\longrightarrow\|$ *denotes the smallest relation induced by the following rule:*

$$\frac{q \xrightarrow{G,\phi} q' \textbf{ and } P \in \|G\|}{q \,\|\xrightarrow{P,\phi}\|\, q'} \tag{8.1}$$

Replacing every synchronization uniconstraint $\{p_1,\ldots,p_n\}$ in a uniconstraint automaton $\boldsymbol{a}$ with synchronization multiconstraint $\{\{\{p_1\},\ldots,\{p_n\}\}\}$ straightforwardly yields a multiconstraint automaton $\boldsymbol{e}$ such that $\boldsymbol{a} = \|\boldsymbol{e}\|$. Henceforth, I tacitly apply this behavior-preserving transformation from uniconstraint automata into multiconstraint automata whenever necessary. If a synchronization multiconstraint matches $\{\{p_1\},\ldots,\{p_n\}\}$ (e.g., every E_i in every synchronization multiconstraint resulting from the previous transformation from uniconstraint automata into multiconstraint automata), I call that E_i *simple*.

Definition 63 (simple choices)**.** $\mathsf{Simpl} : 2^{2^{2^{\mathbb{P}}}} \to 2^{2^{2^{\mathbb{P}}}}$ *denotes the function defined by the following equation:*

$$\mathsf{Simpl}(G) = \{\{\{p_1\}, \ldots, \{p_n\}\} \mid \{\{p_1\}, \ldots, \{p_n\}\} \in G\}$$

If a synchronization multiconstraint has only simple E_i's (i.e., if $\mathsf{Simpl}(G) = G$), I call also that synchronization multiconstraint "simple".

To fire a multitransition in a multiconstraint automaton at run-time, its corresponding protocol unit needs to check that multitransition's synchronization multiconstraint G. To perform such a check, this protocol unit must find some $E_i \in G$ such that for every $V_{ij} \in E_i$, *some* port $p \in V_{ij}$ exists such that its corresponding data structure has a pending I/O operation (i.e., the context of the protocol unit, constituted by its pending I/O operations, must satisfy at least one synchronization uniconstraint in the interpretation of G). If so, the protocol unit can effectuate an instance of interaction involving exactly one port out of every V_{ij} (i.e., corresponding to a synchronization uniconstraint in the interpretation of G). Thus, whenever a protocol unit fires a multitransition in a multiconstraint automaton, it effectively fires a unitransition in the corresponding uniconstraint automaton. Importantly, however, even if an external observer knows that a multitransition fired, without any additional information, this observer cannot possibly know to which particular unitransition this firing corresponds. After all, even if the firing multitransition has only one E_i, the observer cannot know which port out of every V_{ij} the protocol unit selected—the ports in every V_{ij} appear indistinguishable to the observer. Moreover, the protocol unit can efficiently select a port out of every V_{ij} by using a queue for that V_{ij}, as in my manually modified compiler-generated code in the previous subsection. This paragraph, then, establishes the connection between $\big[$queues and port indistinguishability in practice$\big]$ and $\big[$in theory$\big]$.

From a propositional logic perspective, a synchronization multiconstraint

$$G = \{E_1, \ldots, E_n\} = \{\{V_{11}, \ldots, V_{1n_1}\}, \ldots, \{V_{n1}, \ldots, V_{nn_n}\}\}$$

for a multiconstraint automaton with ports P^{all} corresponds to the formula

$$\begin{array}{c} \bigoplus V_{11} \cdots \bigoplus V_{1n_1} \prod \{\overline{p} \mid p \in P^{\text{all}} \textbf{ and } p \notin V_{11} \cup \cdots \cup V_{1n_1}\} \\ + \cdots + \\ \bigoplus V_{n1} \cdots \bigoplus V_{nn_n} \prod \{\overline{p} \mid p \in P^{\text{all}} \textbf{ and } p \notin V_{n1} \cup \cdots \cup V_{nn_n}\} \end{array}$$

where $+$ denotes disjunction, juxtaposition/ $\prod$ denotes conjunction, $\overline{\cdot}$ denotes negation, and $\bigoplus$ denotes uniqueness quantification. With Halle and Arbab, I presented (parts of) the material in this chapter entirely from this propositional logic perspective [JHA14a].

When indeed considered as propositional formulas, synchronization multiconstraints may seem similar to propositional *guards* on transitions in *guarded automata* [BCS09, BCS12], which consist of disjunction, conjunction and negation (but no uniqueness quantification). Bonsangue et al. use such guarded automata for modeling protocols, similar to constraint automata, but significantly

different in expressiveness: guarded automata support modeling *context-sensitive* protocols but not data-sensitive protocols, whereas constraint automata support modeling data-sensitive protocols but not context-sensitive protocols (at least not directly; see my work with Krause and Arbab for an indirect encoding [**J**KA11]). Although synchronization multiconstraints and guards formally may seem similar, they have different applications: guards specify the context in which a transition may—or may not—fire in the work of Bonsangue et al., whereas synchronization multiconstraints specify which ports may participate in a transition, irrespective of context. From a propositional logic perspective, synchronization multiconstraints bear similarities also with the various *connector algebras* of Bliudze and Sifakis [BS08, BS10] and Baranov and Bliudze [BB15], whose propositional formulas over ports—with a richer structure than just disjunction/conjunction/negation—induce sets of sets of ports, to compactly represent sets of synchronization uniconstraints ("interactions" in their terminology). The additional expressive power of these connector algebras, relative to synchronization multiconstraints, makes them suitable for compactly representing a wider range of synchronization patterns (their primary use case) but unsuitable for the kind of manipulation shortly introduced in Definition 67 (the primary use case of synchronization multiconstraints). The idea of labeling transitions with atomic propositions plays a key role also in the translation algorithm from LTL formulas to Büchi automata, in the context of model checking, by Giannakopoulou and Lerda [GL02]. Although atomic propositions significantly differ from propositional formulas in expressive power, interestingly, Giannakopoulou and Lerda have a comparable goal (merging "similar" states into single states to shrink automata) as I have (merging "similar" transitions into single transitions to generalize individual ports into queues of ports). The concept of merging states—but not transitions, to my knowledge—as a means of generalizing models has applications also in (stochastic) automaton/grammar inference [BO05, CO94, LPP98, SO93].

Operations on Multiconstraint Automata

Useful as multiconstraint automata in principle may seem for modeling port indistinguishability and queues, the straightforward transformation from uniconstraint automata into multiconstraint automata, just below Definition 62, does not unleash this full potential quite yet. After all, in multiconstraint automata, port indistinguishability and queues manifest as nonsingleton V_{ij}'s in synchronization multiconstraints, but the synchronization multiconstraints in the multiconstraint automata resulting from that transformation contain only singleton V_{ij}'s. Therefore, I introduce two behavior-preserving operations on such multiconstraint automata. Each of these operations changes the structure of its operand, ultimately to form nonsingleton V_{ij}'s, thereby revealing sets of indistinguishable ports in synchronization multiconstraints.

Recall that my manual modifications to compiler-generated code in the previous subsection essentially correspond to the notion of combining multiple transitions into a single transition. My first operation to change the structure

of multiconstraint automata, therefore, indeed combines multiple multitransitions into a single multitransition. Note that the structure of synchronization uniconstraints in uniconstraint automata does not support such an operation.

Definition 64 (combination)**.** $\mathsf{comb} : \mathbb{A}\text{UTOM}^+ \to \mathbb{A}\text{UTOM}^+$ *denotes the function defined by the following equation:*

$$\mathsf{comb}((Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0))) = (Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow_{\mathsf{comb}}, (q^0, \mu^0))$$

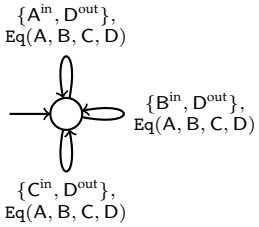
where $\longrightarrow_{\mathsf{comb}}$ *denotes the smallest relation induced by the following rule:*

$$\frac{G^{\text{all}} = \bigcup\{\hat{G} \mid q \xrightarrow{\hat{G},\phi} q'\}}{q \xrightarrow{G^{\text{all}},\phi}_{\mathsf{comb}} q'} \tag{8.2}$$

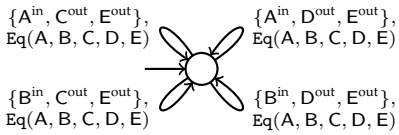
As formalized in Definition 64, to combine multiple multitransitions into a single multitransition, these multitransitions must satisfy three conditions: they must have the same source state, the same target state, and the same data constraint. Especially the latter condition may restrict the extent to which I can combine multitransitions in practice. For instance, in my manual modifications to compiler-generated code in the previous subsection, I effectively combined multitransitions with different—but somehow similar—data constraints (namely A = $x^\bullet$ and B = $x^\bullet$). For now, I simply assume equality of data constraints whenever necessary; I come back to this point in more detail in the next subsection, where I present a basic operation for making "similar" data constraints equal, called *homogenization*.

Figure 8.4 shows uniconstraint automata for two protocols (with homogenized, yet equivalent, data constraints as explained shortly) and their corresponding multiconstraint automata before and after combining their multitransitions; for now, ignore the fourth row with "manipulated synchronization multiconstraints". I introduced the Merger family already in Chapter 3; I introduce the RegulatedMergerRouter family here to demonstrate, shortly, that the optimization technique presented in this chapter supports inference of multiple queues. Figure 8.5 shows a circuit for the same member of the RegulatedMergerRouter$_{2,2}$ subfamily as in Figure 8.4; Figure 8.6 show a FOCAML definition for the entire RegulatedMergerRouter family and a main definition for the same member as in the previous figure. Every member of RegulatedMergerRouter$_{k,l}$ infinitely often atomically [accepts a datum d on one of its k input ports for producers, then offers d both on one of its l output ports for consumers and on another output port for the *regulator*] (where "input" and "output" qualify ports from the protocol perspective). This regulator forms a third party between the group of producers and the group of consumers, regulating (the pace of) flow between those two groups through its `get` operations.

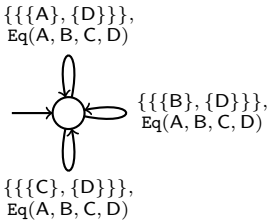
The following theorem states the correctness of Definition 64: combining its multitransitions preserves the interpretation of a multiconstraint automaton.



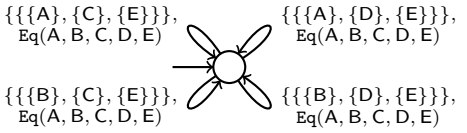
$\mathsf{Merger}_3(\mathsf{A}, \mathsf{B}, \mathsf{C}; \mathsf{D})$: uniconstraint automaton with homogenized data constraints



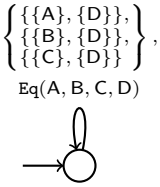
$\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A}, \mathsf{B}; \mathsf{C}, \mathsf{D}, \mathsf{E})$: uniconstraint automaton with homogenized data constraints



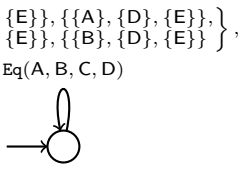
$\mathsf{Merger}_3(\mathsf{A}, \mathsf{B}, \mathsf{C}; \mathsf{D})$: multiconstraint automaton before combining multitransitions



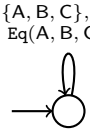
$\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A}, \mathsf{B}; \mathsf{C}, \mathsf{D}, \mathsf{E})$: multiconstraint automaton before combining multitransitions



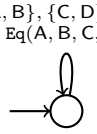
$\mathsf{Merger}_3(\mathsf{A}, \mathsf{B}, \mathsf{C}; \mathsf{D})$: multiconstraint automaton after combining multitransitions



$\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A}, \mathsf{B}; \mathsf{C}, \mathsf{D}, \mathsf{E})$: multiconstraint automaton after combining multitransitions



$\mathsf{Merger}_3(\mathsf{A}, \mathsf{B}, \mathsf{C}; \mathsf{D})$: multiconstraint automaton after combining multitransitions and after manipulating synchronization multiconstraints



$\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A}, \mathsf{B}; \mathsf{C}, \mathsf{D}, \mathsf{E})$: multiconstraint automaton after combining multitransitions and after manipulating synchronization multiconstraints

Figure 8.4: $\mathsf{Merger}_3(\mathsf{A}, \mathsf{B}, \mathsf{C}; \mathsf{D})$ and $\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A}, \mathsf{B}; \mathsf{C}, \mathsf{D}, \mathsf{E})$

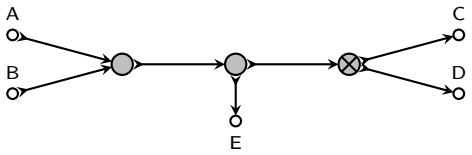


Figure 8.5: Circuit for a member of subfamily RegulatedMergerRouter$_{2,2}$

```
1  RegulatedMergerRouter(in[];out[],out_regulator) = {
2    { prod i:1..#in { Sync(in[i];P1[i]) } }
3      mult Merger(P1[1..#in];P2)
4        mult Replicator2(P2;P3,P6)
5          mult Sync(P3;P4)
6            mult Router(P4,P5[1..#out])
7              mult { prod i:1..#out { Sync(P5[i];out[i]) } }
8          mult Sync(P6;out_regulator)
9  }

10 main = { RegulatedMergerRouter([A,B];[C,D],E) }
```

Figure 8.6: FOCAML definition for family RegulatedMergerRouter and a main definition for a member of RegulatedMergerRouter$_{2,2}$

Theorem 22. $\|e\| = \|\mathsf{comb}(e)\|$

Once multiconstraint automata have combined multitransitions (i.e., after applying comb), for each of their synchronization multiconstraints, I can compute maximal sets of indistinguishable ports. I consider ports indistinguishable if they occur in exactly the same E_i's "modulo occurrences of those ports". To understand the latter phrase, consider the following example:

$$G = \{\underbrace{\{\overbrace{\{\mathsf{A}\}}^{V_{11}}, \overbrace{\{\mathsf{D}\}}^{V_{12}}\}}_{E_1}, \underbrace{\{\overbrace{\{\mathsf{B}\}}^{V_{21}}, \overbrace{\{\mathsf{D}\}}^{V_{22}}\}}_{E_2}, \underbrace{\{\overbrace{\{\mathsf{C}\}}^{V_{31}}, \overbrace{\{\mathsf{D}\}}^{V_{32}}\}}_{E_3}\}$$

Under this G, I can equate E_1, E_2, and E_3 to each other modulo the occurrence of ports A, B, and C, because by removing $\{\mathsf{A}\}$, $\{\mathsf{B}\}$, and $\{\mathsf{C}\}$ from E_1, E_2, and E_3, I get exactly the same *remainder* $\{\{\mathsf{D}\}\}$. Intuitively, this notion of port indistinguishability formalizes the idea that even if an external observer knows that the ports in the remainder of $E_i \in G$ participated in the firing of a G-labeled multitransition (e.g., D), this observer cannot know which port additionally participated in that firing (e.g., A, B, or C)—the ports that additionally may have participated remain indistinguishable to the observer. Let $\mathsf{Port}(G)$ denote the ports that occur in a synchronization multiconstraint G.

Definition 65 (remainders)**.** $\mathsf{Remaind} : 2^{2^{2^{\mathbb{P}}}} \times \mathbb{P} \to 2^{2^{2^{\mathbb{P}}}}$ *denotes the function defined by the following equation:*

$$\mathsf{Remaind}_G(p) = \{E \setminus \{V\} \mid p \in V \in E \in G\}$$

Definition 66 (indistinguishability)**.** $\mathsf{Indist} : 2^{2^{2^{\mathbb{P}}}} \to 2^{2^{\mathbb{P}}}$ *denotes the function defined by the following equation:*

$$\mathsf{Indist}(G) = \left\{ P \,\middle|\, \begin{array}{l} [P = \{p \mid p \in \mathsf{Port}(G) \ \textbf{and} \ \mathsf{Remaind}_G(p) = G'\} \ \textbf{for some} \ G'] \\ \textbf{and} \ P \neq \emptyset \end{array} \right\}$$

Lemma 19. $\mathsf{Indist}(G)$ *denotes a partition of* $\mathsf{Port}(G)$

For instance, let $G = \{\{\{\mathsf{A}\},\{\mathsf{D}\}\},\{\{\mathsf{B}\},\{\mathsf{D}\}\},\{\{\mathsf{C}\},\{\mathsf{D}\}\}\}$ (i.e., the synchronization multiconstraint in $\mathsf{Merger}_3(\mathsf{A},\mathsf{B},\mathsf{C};\mathsf{D})$ in Figure 8.4, third row). Then:

$$\begin{array}{ll} \mathsf{Remaind}_G(\mathsf{A}) & = \{\{\{\mathsf{D}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{B}) & = \{\{\{\mathsf{D}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{C}) & = \{\{\{\mathsf{D}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{D}) & = \{\{\{\mathsf{A}\}\},\{\{\mathsf{B}\}\},\{\{\mathsf{C}\}\}\} \\ \mathsf{Indist}(G) & = \{\{\mathsf{A},\mathsf{B},\mathsf{C}\},\{\mathsf{D}\}\} \end{array}$$

As a more complex example, let

$$G = \{\{\{\mathsf{A}\},\{\mathsf{C}\},\{\mathsf{E}\}\},\{\{\mathsf{A}\},\{\mathsf{D}\},\{\mathsf{E}\}\},\{\{\mathsf{B}\},\{\mathsf{C}\},\{\mathsf{E}\}\},\{\{\mathsf{B}\},\{\mathsf{D}\},\{\mathsf{E}\}\}\}$$

(i.e., the synchronization multiconstraint in $\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A},\mathsf{B};\mathsf{C},\mathsf{D},\mathsf{E})$ in Figure 8.4, third row). Then:

$$\begin{array}{ll} \mathsf{Remaind}_G(\mathsf{A}) & = \{\{\{\mathsf{C}\},\{\mathsf{E}\}\},\{\{\mathsf{D}\},\{\mathsf{E}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{B}) & = \{\{\{\mathsf{C}\},\{\mathsf{E}\}\},\{\{\mathsf{D}\},\{\mathsf{E}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{C}) & = \{\{\{\mathsf{A}\},\{\mathsf{E}\}\},\{\{\mathsf{B}\},\{\mathsf{E}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{D}) & = \{\{\{\mathsf{A}\},\{\mathsf{E}\}\},\{\{\mathsf{B}\},\{\mathsf{E}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{E}) & = \{\{\{\mathsf{A}\},\{\mathsf{C}\}\},\{\{\mathsf{A}\},\{\mathsf{D}\}\},\{\{\mathsf{B}\},\{\mathsf{C}\}\},\{\{\mathsf{B}\},\{\mathsf{D}\}\}\} \\ \mathsf{Indist}(G) & = \{\{\mathsf{A},\mathsf{B}\},\{\mathsf{C},\mathsf{D}\},\{\mathsf{E}\}\} \end{array}$$

In this more complex example, thus, I get two nonsingleton V_{ij}'s, each of which corresponds to a different queue.

Using the previous formalization of port indistinguishability, I define an operation for manipulating simple synchronization multiconstraints such that afterward, every V_{ij} corresponds to a set of indistinguishable ports. The restriction to simple synchronization multiconstraint does not affect the applicability of this manipulation (in the current context), because a FOCAML compiler manipulates only multiconstraint automata derived from uniconstraint automata using the transformation below Definition 60; such multiconstraint automata have only simple synchronization multiconstraints.

Definition 67 (manipulation of synchronization multiconstraints)**.** $\wr\cdot\int : 2^{2^{2^{\mathbb{P}}}} \to 2^{2^{2^{\mathbb{P}}}}$ *denotes the function defined by the following equation:*

$$\wr G\int = \begin{cases} \left\{ \{P_1,\ldots,P_n\} \;\middle|\; \begin{array}{l} \{p_1,\ldots,p_n\} \in \|G\| \\ \textbf{and}\ \ p_1 \in P_1 \in \mathsf{Indist}(G) \\ \textbf{and}\ \ \cdots \\ \textbf{and}\ \ p_n \in P_n \in \mathsf{Indist}(G) \end{array} \right\} & \textbf{if}\ \ G = \mathsf{Simpl}(G) \\ G & \textbf{otherwise} \end{cases}$$

The following lemma states the correctness of Definition 67: manipulating a synchronization multiconstraint preserves the interpretation of that synchronization multiconstraint.

Lemma 20. $\|G\| = \|\wr G\int\|$

Next, I straightforwardly extend manipulation of synchronization multiconstraints to multiconstraint automata.

Definition 68 (manipulation of multiconstraint automata)**.** $\wr\cdot\int : \mathbb{A}\text{UTOM}^{+} \to \mathbb{A}\text{UTOM}^{+}$ *denotes the function defined by the following equation:*

$$\begin{array}{c} \wr(Q,(P^{\text{all}},P^{\text{in}},P^{\text{out}}),M,\longrightarrow,(q^0,\mu^0))\int = \\ (Q,(P^{\text{all}},P^{\text{in}},P^{\text{out}}),M,\wr\longrightarrow\int,(q^0,\mu^0)) \end{array}$$

where $\wr\longrightarrow\int$ *denotes the smallest relation induced by the following rule:*

$$\frac{q \xrightarrow{G,\phi} q'}{q \wr\xrightarrow{\wr G\int,\phi}\int q'} \tag{8.3}$$

By manipulating the multiconstraint automata on the third row in Figure 8.4 according to Definition 68, I get the multiconstraint automata on the fourth row in the same figure. These resulting multiconstraint automata have synchronization multiconstraints with nonsingleton V_{ij}'s—as desired—and thereby make the indistinguishability of the ports in those V_{ij}'s structurally explicit. Manipulation of synchronization multiconstraints in multiconstraint automata thus enables a FOCAML compiler to automatically identify indistinguishable ports to infer queues and generate queue-optimized code. Note that for $\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A},\mathsf{B};\mathsf{C},\mathsf{D},\mathsf{E})$ in Figure 8.4, doing so yields two queues instead of just one.

The following theorem states the correctness of Definition 68: manipulating its synchronization multiconstraints preserves the interpretation of a multiconstraint automaton.

Theorem 23. $\|\boldsymbol{e}\| = \|\wr\boldsymbol{e}\int\|$

Figure 8.7 shows the compilation approach resulting from queue-inference as just formalized (in which I leave the straightforward step to transform uniconstraint automata $\boldsymbol{d}_1,\ldots,\boldsymbol{d}_m$ into multiconstraint automata implicit).

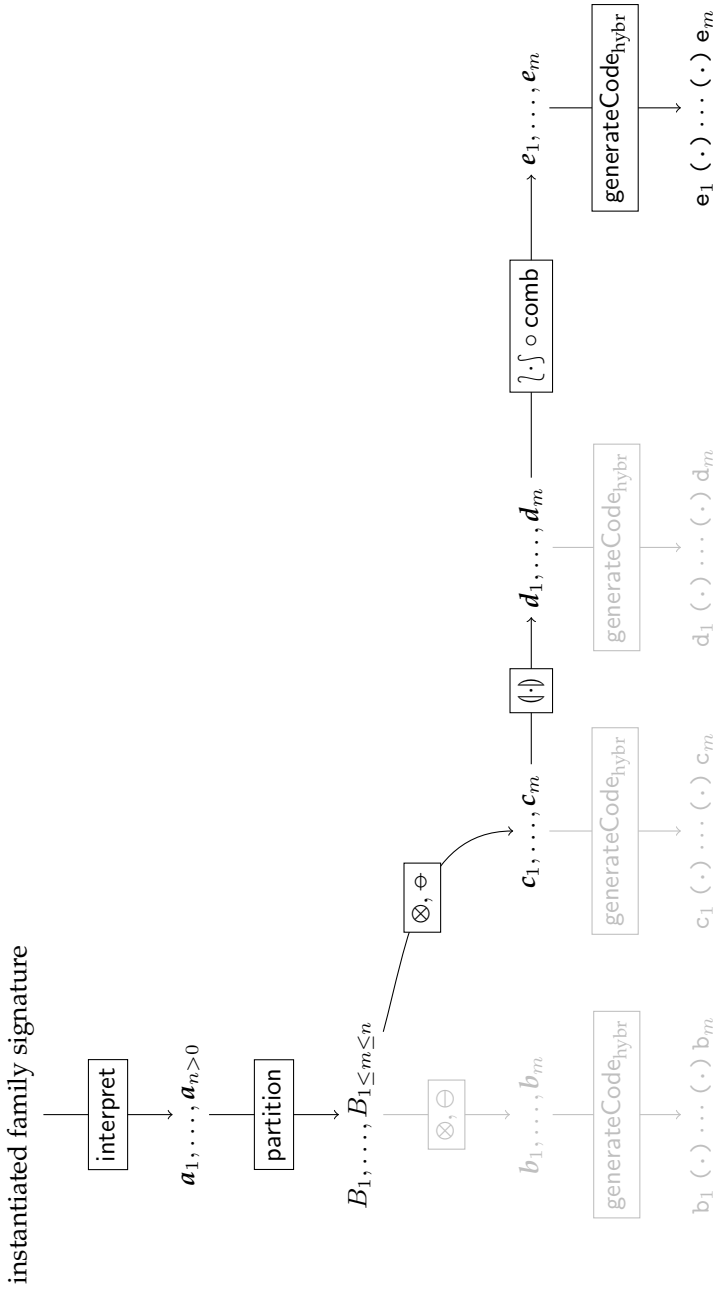


Figure 8.7: Hybrid compilation approach with syntactic subtraction, commandification, and queue-inference

$$\frac{\left[p_1, p_2 \in P \cap \mathrm{Dom}(\sigma) \ \textbf{implies} \ \sigma \models p_1 = p_2\right] \ \textbf{for all} \ p_1, p_2}{\sigma \models \mathtt{Eq}(P)} \tag{8.4}$$

Figure 8.8: Addendum to Definition 69

Homogenization

To combine multiple multitransitions into a single multitransition, by Definition 64 of comb, these multitransitions must have exactly the same data constraint. So far, I simply assumed that this condition holds true, but in fact, all practical cases where queue-inference may improve performance that I know of violate this condition. For instance, Merger_3 in Figure 8.4, first row, actually has A = D, B = D, and C = D as its data constraints. This makes their corresponding multitransitions not amenable for combination. To solve this problem, I must *homogenize* "similar" data constraints before trying to combine their corresponding multitransitions, the result of which Figure 8.4 already exemplifies. In this subsection, I present one approach to such homogenization. Because I apply homogenization only to uniconstraint automata (i.e., before translating them into multiconstraint automata), henceforth, I simply write "constraint automaton", "transition", and "synchronization constraint" (instead of "uniconstraint automaton", "unitransition", and "synchronization uniconstraint").

Essentially, my basic approach to homogenization presented in the rest of this section replaces conjunctions of $p_1 = p_2$ data equalities with a new kind of data atom, $\mathtt{Eq}(P)$. Informally, $\mathtt{Eq}(P)$ means that every port in P has either the same value or no value. Let $\mathbb{DC}^{\mathtt{Eq}}$ denote $\mathbb{DC}$ extended with $\mathtt{Eq}(P)$ data atoms, and let $\mathbb{DC}^{\mathtt{Eq}}_{\exists,\wedge}$ denote the corresponding set of normal data constraints. I define a new entailment relation for $\mathbb{DC}^{\mathtt{Eq}}$.

Definition 69 (entailment with Eq)**.** $\models \subseteq \mathbb{ASSIGNM} \times \mathbb{DC}^{\mathtt{Eq}}$ *denotes the smallest relation induced by the rules in Definition 16 and Figure 8.8.*

Thus, in the rest of this section, I overload $\models$ from the previous Definition 16 with the current Definition 69. Similarly, let $\Rightarrow$ and $\equiv$ denote the implication and the equivalence relation derived from $\models$ in Definition 69 (instead of from $\models$ in Definition 16) in the usual way for first-order logic [Rau10a]. Finally, in the rest of this section, let every data constraint in every constraint automaton come from $\mathbb{DC}^{\mathtt{Eq}}$ instead of from $\mathbb{DC}$. Shortly, I explain the significant semantic difference between $\mathtt{Eq}(P)$ and enumerations of data terms for equating the ports in P in more detail.

To first more informally explain the process of homogenization, suppose that I have a transition (q, P, ϕ, q') in a constraint automaton with ports P^{all} (i.e., $P \subseteq P^{\mathrm{all}}$). Also, suppose that ϕ contains no data variables for memory cells (i.e., $\mathsf{Free}(\phi) \subseteq P$). Finally, suppose that ϕ consist of sufficiently many

data equalities to equate—either directly or transitively—all ports in P to each other (i.e., $P \subseteq \mathsf{Free}(\phi)$). Under these assumptions, clearly, I can safely replace ϕ with $\mathtt{Eq}(P)$. Less than clearly, however, I can also safely replace ϕ with $\mathtt{Eq}(P^{\text{all}})$. After all, whenever transition (q, P, ϕ, q') fires, its corresponding data assignment σ has exactly the ports in P in its domain and no other ports (i.e., $\mathrm{Dom}(\sigma) = P$). Consequently, the condition $p_1, p_2 \in P^{\text{all}} \cap \mathrm{Dom}(\sigma)$ in the premise of Rule 8.4 in Figure 8.8 (instantiated for P^{all}) reduces to $p_1, p_2 \in P^{\text{all}} \cap P$, which in turn reduces to $p_1, p_2 \in P$. In other words, when I evaluate $\mathtt{Eq}(P^{\text{all}})$ in the context of transition (q, P, ϕ, q'), its semantics causes Eq to ignore all ports in $P^{\text{all}} \setminus P$. In those cases, thus, $\mathtt{Eq}(P^{\text{all}})$ reduces to $\mathtt{Eq}(P)$. To homogenize data constraints in a constraint automaton, then, I substitute $\mathtt{Eq}(P^{\text{all}})$ for every data constraint with similar properties as the previous ϕ. For instance, in $\mathsf{Merger}_3(\mathsf{A}, \mathsf{B}, \mathsf{C}; \mathsf{D})$, with transitions $(q, \{\mathsf{A}, \mathsf{D}\}, \mathsf{A} = \mathsf{D}, q)$, $(q, \{\mathsf{B}, \mathsf{D}\}, \mathsf{B} = \mathsf{D}, q)$, and $(q, \{\mathsf{C}, \mathsf{D}\}, \mathsf{C} = \mathsf{D}, q)$, every data constraint has the required properties, and therefore, homogenization replaces each of them with $\mathtt{Eq}(\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\})$. Subsequently, I can translate the resulting homogenized constraint automaton into a multiconstraint automaton as explained in the previous subsection and exemplified in Figure 8.4.

To formalize homogenization, I restrict myself to normalized constraint automata (without loss of generality). To determine, then, whether a normal data constraint φ has sufficiently many data equalities to equate all ports in a synchronization constraint P to each other, let function EqTerm construct the edge relation of a graph, whose every vertex corresponds to a data term in φ, and whose every edge corresponds to a data equality in φ.

Definition 70 (equated data terms)**.** $\mathsf{EqTerm} : \mathbb{DC}^{\mathtt{Eq}}_{\exists,\wedge} \to 2^{\mathbb{TERM} \times \mathbb{TERM}}$ *denotes the function defined by the following equations:*

$$
\begin{aligned}
\mathsf{EqTerm}(a) &= \begin{cases} \{(t_1, t_2), (t_2, t_1)\} & \textbf{if } a = t_1 = t_2 \\ \emptyset & \textbf{otherwise} \end{cases} \\
\mathsf{EqTerm}(\neg a) &= \emptyset \\
\mathsf{EqTerm}(\exists x.\varphi) &= \mathsf{EqTerm}(\varphi) \\
\mathsf{EqTerm}(\ell_1, \ldots, \ell_k) &= \mathsf{EqTerm}(\ell_1) \cup \cdots \cup \mathsf{EqTerm}(\ell_k)
\end{aligned}
$$

For a normal data constraint φ to have sufficiently many data equalities to equate all ports in P to each other, then, a path from every $p_1 \in P$ to every other $p_2 \in P$ must exist in $\mathsf{EqTerm}(\varphi)$. Equivalently, the transitive closure of $\mathsf{EqTerm}(\varphi)$, denoted by $\mathsf{EqTerm}(\varphi)^+$, must contain the pair (p_1, p_2) for all $p_1, p_2 \in P$. Furthermore, as previously explained, φ must not contain memory cell variables, and for simplicity, I also forbid the occurrence of data functions and data relations. Let $\mathsf{Term}(\varphi)$ denote the set of data terms that occur in φ, and let $\mathsf{OnlyEq}(\varphi)$ hold true iff φ contains only data equalities (and no data relations).

Definition 71 (total data equalities over sets of ports)**.** $\mathsf{Eq} : 2^{\mathbb{P}} \to 2^{\mathbb{DC}^{\mathsf{Eq}}_{\exists,\wedge}}$ *denotes the function defined by the following equation:*

$$\mathsf{Eq}(P) = \left\{ \varphi \;\middle|\; \begin{array}{l} \{(p_1, p_2) \mid p_1, p_2 \in P\} \subseteq \mathsf{EqTerm}(\varphi)^+ \\ \textbf{and } \mathsf{Term}(\varphi) = \mathsf{Free}(\varphi) = P \\ \textbf{and } \mathsf{OnlyEq}(\varphi) \end{array} \right\}$$

I define homogenization in terms of Eq and $\mathtt{Eq}$.

Definition 72 (homogenization)**.** $\langle\cdot\rangle : \mathbb{A}\mathrm{UTOM} \to \mathbb{A}\mathrm{UTOM}$ *denotes the function defined by the following equation:*

$$\begin{array}{c} \langle (Q, (P^{\mathrm{all}}, P^{\mathrm{in}}, P^{\mathrm{out}}), M, \longrightarrow, (q^0, \mu^0)) \rangle = \\ (Q, (P^{\mathrm{all}}, P^{\mathrm{in}}, P^{\mathrm{out}}), M, \langle\longrightarrow\rangle, (q^0, \mu^0)) \end{array}$$

where $\langle\longrightarrow\rangle$ *denotes the smallest relation induced by the following rules:*

$$\frac{q \xrightarrow{P,\phi} q' \;\textbf{ and }\; \phi \in \mathsf{Eq}(P)}{q \;\langle\xrightarrow{P,\mathsf{Eq}(P^{\mathrm{all}})}\rangle\; q'} \quad (8.5) \qquad \frac{q \xrightarrow{P,\phi} q' \;\textbf{ and }\; \phi \notin \mathsf{Eq}(P)}{q \;\langle\xrightarrow{P,\phi}\rangle\; q'} \quad (8.6)$$

Problematically, if I substitute $\mathtt{Eq}(P^{\mathrm{all}})$ for ϕ in a transition (q, P, ϕ, q'), I do not preserve the semantics of ϕ in the sense of $\equiv$ (i.e., the equivalence relation on data constraints derived from $\models$): although satisfaction of $\mathtt{Eq}(P^{\mathrm{all}})$ implies satisfaction of ϕ (i.e., $\mathtt{Eq}(P^{\mathrm{all}}) \Rightarrow \phi$), satisfaction of ϕ does not imply satisfaction of $\mathtt{Eq}(P^{\mathrm{all}})$ (i.e., $\phi \not\Rightarrow \mathtt{Eq}(P^{\mathrm{all}})$). For instance, whereas data assignment $\{\mathsf{A} \mapsto 0, \mathsf{B} \mapsto 0, \mathsf{C} \mapsto 0, \mathsf{D} \mapsto 0\}$ satisfies both $\mathtt{Eq}(\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\})$ and $\mathsf{A} = \mathsf{D}$, data assignment $\{\mathsf{A} \mapsto 0, \mathsf{B} \mapsto 1, \mathsf{D} \mapsto 0\}$ satisfies $\mathsf{A} = \mathsf{D}$ but not $\mathtt{Eq}(\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\})$. To prove the correctness of Definition 72, therefore, I need to develop some more technical machinery, based on a *tighter* notion of implication/equivalence for data constraints and their corresponding notion of behavioral preorder/congruence for constraint automata.

Definition 73 (tight implication)**.** $\Rightarrow^{\mathsf{t}} \subseteq 2^{\mathbb{P}} \times \mathbb{DC}^{\mathsf{Eq}} \times \mathbb{DC}^{\mathsf{Eq}}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\left[\begin{bmatrix} P = \mathbb{P} \cap \mathrm{Dom}(\sigma) \\ \textbf{and } \sigma \models \phi_1 \end{bmatrix} \textbf{ implies } \sigma \models \phi_2\right] \textbf{ for all } \sigma}{\phi_1 \Rightarrow^{\mathsf{t}}_P \phi_2} \quad (8.7)$$

Definition 74 (tight equivalence)**.** $\equiv^{\mathsf{t}} \subseteq 2^{\mathbb{P}} \times \mathbb{DC}^{\mathsf{Eq}} \times \mathbb{DC}^{\mathsf{Eq}}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\phi_1 \Rightarrow^{\mathsf{t}}_P \phi_2 \;\textbf{ and }\; \phi_2 \Rightarrow^{\mathsf{t}}_P \phi_1}{\phi_1 \equiv^{\mathsf{t}}_P \phi_2} \quad (8.8)$$

Revisiting my previous example, $\mathtt{Eq}(\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\})$ tightly implies $\mathsf{A} = \mathsf{D}$ under any set of ports (because implication subsumes tight implication), while $\mathsf{A} = \mathsf{D}$

tightly implies $\mathsf{Eq}(\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{D}\})$ under $\{\mathsf{A},\mathsf{D}\}$. Thus, $\mathsf{Eq}(\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{D}\})$ and $\mathsf{A} = \mathsf{D}$ have tightly equivalent semantics under $\{\mathsf{A},\mathsf{D}\}$. The following lemma generalizes this example: it states that whenever a data constraint contains sufficiently many data equalities to equate all ports in P to each other, this data constraint and data constraint $\mathsf{Eq}(P^{\text{all}})$ have tightly equivalent semantics under P.

Lemma 21. $\left[\phi \in \mathsf{Eq}(P) \text{ and } P \subseteq P^{\text{all}}\right]$ **implies** $\phi \equiv^{\text{t}}_{P} \mathsf{Eq}(P^{\text{all}})$

Using tight implication, I can formulate a tighter version of the behavioral preorder and the behavioral congruence in Definitions 24 and 25. Under these tighter versions—and as the only difference with respect to Definitions 24 and 25—for one constraint automaton to "tightly simulate" a transition of another constraint automaton, the data constraint of the simulated transition must tightly imply the data constraint on the simulating transition under the synchronization constraint of the simulated transition.

Definition 75 (tight behavioral preorder). $\preceq^{\text{t}} \subseteq 2^{\mathbb{Q}\times\mathbb{Q}} \times \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\begin{array}{l} R \subseteq Q_1 \times Q_2 \text{ and } q_1^0 \; R \; q_2^0 \\ \text{and } \left[\begin{array}{c} \left[\, \left[\begin{array}{l} q_1 \xrightarrow{P,\phi_1}_1 q_1' \\ \text{and } q_1 \; R \; q_2 \end{array}\right] \text{ implies } \phi_1 \Rightarrow^{\text{t}}_{P} \bigvee \left\{ \phi_2 \;\middle|\; \begin{array}{l} q_2 \xrightarrow{P,\phi_2}_2 q_2' \\ \text{and } q_1' \; R \; q_2' \end{array} \right\} \right] \\ \text{for all } q_1, q_1', q_2, P, \phi_1 \end{array}\right] \end{array}}{\begin{array}{c} (Q_1, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow_1, (q_1^0, \mu^0)) \\ \preceq^{\text{t}}_{R} (Q_2, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow_2, (q_2^0, \mu^0)) \end{array}} \tag{8.9}$$

Definition 76 (tight behavioral congruence). $\simeq^{\text{t}} \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\left[\boldsymbol{a}_1 \preceq^{\text{t}}_{R} \boldsymbol{a}_2 \text{ and } \boldsymbol{a}_2 \preceq^{\text{t}}_{R^{-1}} \boldsymbol{a}_1\right] \text{ for some } R}{\boldsymbol{a}_1 \simeq^{\text{t}} \boldsymbol{a}_2} \tag{8.10}$$

Behavioral congruence subsumes tight behavioral congruence. Moreover, the following theorems state that tight behavioral congruence implies behavioral equivalence and that tight behavioral congruence constitutes a congruence under the multiplications previously defined in this thesis.

Theorem 24. $\boldsymbol{a}_1 \simeq^{\text{t}} \boldsymbol{a}_2$ **implies** $\boldsymbol{a}_1 \approx \boldsymbol{a}_2$

Theorem 25. $\left[\begin{array}{l} \boldsymbol{a}_1 \circledast_* \boldsymbol{a}_3, \boldsymbol{a}_2 \circledast_* \boldsymbol{a}_4 \in \mathbb{A}\text{UTOM} \\ \text{and } \boldsymbol{a}_1 \simeq^{\text{t}} \boldsymbol{a}_2 \text{ and } \boldsymbol{a}_3 \simeq^{\text{t}} \boldsymbol{a}_4 \end{array}\right]$ **implies** $\boldsymbol{a}_1 \circledast_* \boldsymbol{a}_3 \simeq^{\text{t}} \boldsymbol{a}_2 \circledast_* \boldsymbol{a}_4$

Theorem 26. $\left[\begin{array}{l} \boldsymbol{a}_1 \otimes \boldsymbol{a}_3, \boldsymbol{a}_2 \otimes \boldsymbol{a}_4 \in \mathbb{A}\text{UTOM} \\ \text{and } \boldsymbol{a}_1 \simeq^{\text{t}} \boldsymbol{a}_2 \text{ and } \boldsymbol{a}_3 \simeq^{\text{t}} \boldsymbol{a}_4 \end{array}\right]$ **implies** $\boldsymbol{a}_1 \otimes \boldsymbol{a}_3 \simeq^{\text{t}} \boldsymbol{a}_2 \otimes \boldsymbol{a}_4$

Theorem 27. $\begin{bmatrix} \boldsymbol{a}_1 \odot \boldsymbol{a}_3, \boldsymbol{a}_2 \odot \boldsymbol{a}_4 \in \mathbb{A}\text{UTOM} \\ \textbf{and } \boldsymbol{a}_1 \simeq^{\text{t}} \boldsymbol{a}_2 \textbf{ and } \boldsymbol{a}_3 \simeq^{\text{t}} \boldsymbol{a}_4 \end{bmatrix}$ **implies** $\boldsymbol{a}_1 \odot \boldsymbol{a}_3 \simeq^{\text{t}} \boldsymbol{a}_2 \odot \boldsymbol{a}_4$

Using all this tight machinery, I can finally prove the correctness of Definition 72 of $\langle\cdot\rangle$: homogenizing a constraint automaton preserves its behavior.

Theorem 28. $\boldsymbol{a} \simeq^{\text{t}} \langle \boldsymbol{a} \rangle$

The use of tight behavioral congruence instead of behavioral congruence in the previous theorem remains inconsequential, because only behavioral equivalence truly matters in the end (i.e., accepted interaction languages), which Theorem 24 guarantees for tightly behaviorally congruent constraint automata. Furthermore, because $\simeq^{\text{t}}$ denotes a congruence under $\odot$ by Theorem 27, I can safely replace every constraint automaton with its homogenized version in an l-multiplication expression (i.e., $\boldsymbol{d}_1 \odot \cdots \odot \boldsymbol{d}_m \simeq^{\text{t}} \langle \boldsymbol{d}_1 \rangle \odot \cdots \odot \langle \boldsymbol{d}_m \rangle$ in Figure 8.7).

Homogenization as presented above has limitations, notably as it does not support data functions and data relations. The development of a more general approach to homogenization of data constraints seems an interesting—and challenging—piece of future work. One possible approach to such generalized homogenization consists of the introduction of a "metaquantifier" whose "metavariables" range over data variables (cf. $\exists$, whose data variables range over data). Consider, for instance, transitions $(q, \{\mathsf{A}, \mathsf{C}\}, \texttt{incr}(\mathsf{A}) = \mathsf{C}, q')$ and $(q, \{\mathsf{B}, \mathsf{C}\}, \texttt{incr}(\mathsf{B}) = \mathsf{C}, q')$. Homogenization as defined in Definition 72 leaves these transitions untouched, thereby inhibiting their combination later on. The metaquantifier-based homogenization that I imagine, in contrast, replaces the data constraints in these transitions with $\mathcal{M}[\xi : \{\mathsf{A}, \mathsf{B}\}].\texttt{incr}(\xi) = \mathsf{C}$, where $\mathcal{M}$ denotes the metaquantifier, where ξ denotes a metavariable that ranges over $\{\mathsf{A}, \mathsf{B}\}$, and where $\texttt{incr}(\xi) = \mathsf{C}$ denotes a *template* (i.e., a data constraint with metavariables at places where normally data variables occur). Metaquantification, then, has the following semantics:

$$\sigma \models \mathcal{M}[\xi : X].\theta \textbf{ iff } \left[\left[x \in X \textbf{ and } \sigma \models \theta\{x/\xi\}\right] \textbf{ for some } x\right]$$

where θ denotes a template and where $\theta\{x/\xi\}$ denotes the syntactic substitution of data variable x for metavariable ξ. The main challenge with this approach lies in the inference of metaquantifications, for which I have not found an elegant solution yet.

8.2 Practice

(I have not yet submitted material in this section for publication.)

Compiler

I extended Lykos with the ability to apply queue-inference as in Figure 8.7, controllable through flag `INFER_QUEUES`. When raised, after having computed

product automata (and after subtraction of internal ports, either semantically or syntactically depending on the status of the SUBTRACT_SYNTACTICALLY flag), Lykos homogenizes data constraints, translates constraint automata into multiconstraint automata, merges transitions with homogenized data constraints, and modifies the synchronization multiconstraints on the resulting multitransitions to infer indistinguishable ports, all as described in Section 8.1.

To avoid unnecessary queue overhead, Lykos does not inject queues for singleton sets of indistinguishable ports or for sets with ports of mixed polarity. Also, Lykos injects queues only for ports that occupy the same set of indistinguishable ports in every multitransition. For instance, if one multitransition has a synchronization multiconstraint with $V_{ij} = \{\mathsf{A}, \mathsf{B}\}$, while another multitransition has a synchronization multiconstraint with $V_{ij} = \{\mathsf{A}, \mathsf{C}\}$, Lykos injects a queue neither for A and B nor for A and C. Otherwise, whenever a thread performs an I/O operation on the data structure for A, it has to offer this data structure into both queues, which subsequently requires additional machinery to keep the queues consistent (e.g., whenever another thread polls the data structure for A from one of the queues, it must remove this data structure also from the other queue). Not injecting queues in these cases avoids this kind of overhead. More advanced schemes for injecting queues may exist, though, that alleviate or completely eliminate such overhead. One possible such scheme, for instance, comprises the organization of queues in *conjunctions*, or more generally, in BDD-like structures, by combining unitransitions into (more advanced than in this chapter) multitransitions to further reduce the cost of checking their enabledness. Investigating such schemes seems an interesting opportunity for future work.

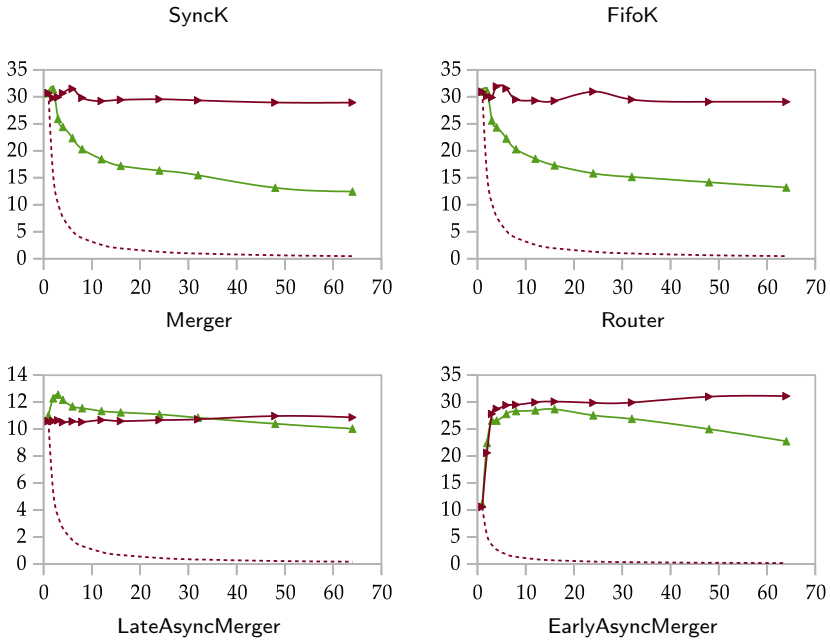
In contrast to Lykos's internals, the run-time library requires no significant modifications, beside the addition of a queue data structure. I implemented this data structure as a concurrent circular buffer, using a semaphore to control concurrent accesses. At run-time, every instance of this data structure has a capacity n, where n equals the size of the set of indistinguishable ports to which this instance corresponds. Code generated with queue-inference differs somewhat from code generated without queue-inference, primarily to account for queues, as already explained in Section 8.1.

Experiments I: Protocols

I repeated the same experiments as in Chapter 7 (and Chapters 4, 5, and 6), generating code for members of families SyncK, FifoK, Merger, Router, LateAsyncMerger, EarlyAsyncMerger, OddFibonacci, and Chess with the INFER_QUEUES-flag raised, but otherwise under the same conditions as in Chapter 7. Figure 8.9 shows the per-family experimental results, averaged over five runs. The solid lines represent the actual measurements; the dotted lines represent inverse-proportional growth with respect to $k = 1$. The purple lines represent the new results; the green lines represent the results from Chapter 6. For SyncK, FifoK, OddFibonacci, and Chess, Lykos generated exactly the same code as in Chapter 7—members of these families have no indistinguishable ports to optimize.

—same as in Figure 7.10—

—same as in Figure 7.10—



—same as in Figure 7.10—

—same as in Figure 7.10—

OddFibonacci

Chess

Figure 8.9: Performance (in number of completed rounds per four minutes) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

In contrast, for all members of Merger, Router, LateAsyncMerger, and EarlyAsyncMerger, queue-inference has the desirable effect that the performance or their generated code becomes constant in the number of producers/consumers.

Figure 8.10 shows per-family speedup charts corresponding to the measurements in Figure 8.9; the dotted lines represent equal performance. Only for LateAsyncMergers and smaller values of k, code generated without queue-inference outperforms code generated with queue-inference. In these cases, the overhead of managing queues outweighs their benefits. In all other cases, code generated with queue-inference outperforms code generated without queue-inference, by an increasing margin in k, up to a speedup of 133% for Mergers, of 120% for Routers, of 8% for LateAsyncMergers, and of 37% for EarlyAsyncMergers.

Experiments II: Programs

I repeated the same experiments as in Chapter 7 (and Chapters 4, 5 and 6), generating code for the NPB benchmarks with the `INFER_QUEUES`-flag raised, but otherwise under the same conditions as in Chapter 7.

Figures 8.11–8.18 show performance charts for the FOCAML-to-Java-compiled versions of the NPB kernel benchmarks (averaged over five runs), speedup charts (with respect to their Java versions by Frumkin et al.), and charts about cache misses. In the absence of dotted purple lines, the dotted green lines represent the MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks without queue-inference; the solid purple/green lines represent the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions; the dotted black lines represent the Java versions by Frumkin et al. For the MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions, Lykos generated exactly the same code as in Chapter 7—members of these families have no indistinguishable ports to optimize—and therefore, I have no new results for these versions (i.e., Figures 8.11–8.18 have no dotted purple lines).

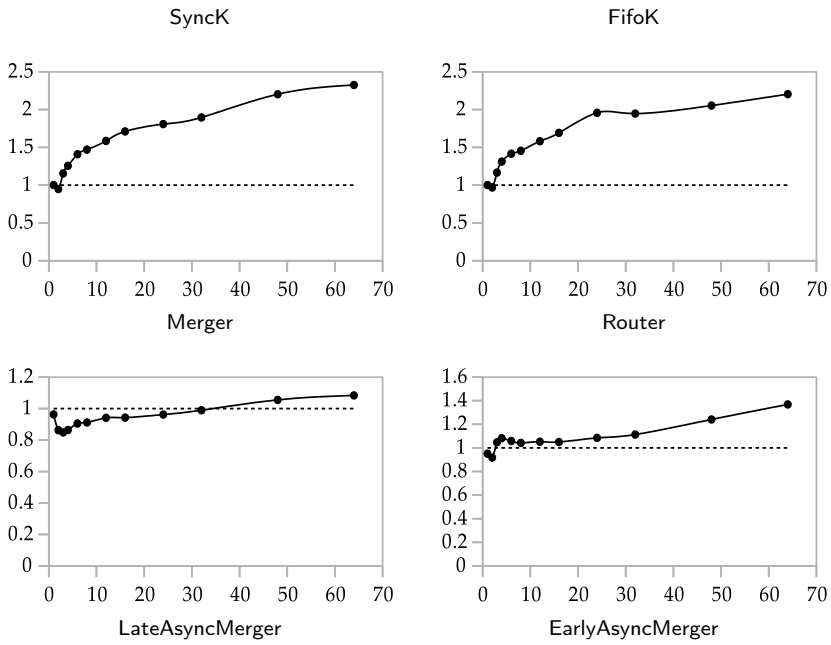
I make the following main observations about these experimental results:

- Overall, the FOCAML-to-Java-compiled versions of the NPB kernel benchmarks with queue-inference outperform the FOCAML-to-Java-compiled versions without queue-inference (purple lines versus green lines).

- Overall, the Java versions of the NPB kernel benchmarks by Frumkin et al. and the FOCAML-to-Java-compiled versions with queue-inference have roughly similar performance: in some cases the former outperform the latter, while in other cases, the latter outperform the former (e.g., in NPB-FT for smaller values of k; in NPB-MG and NPB-CG for larger values of k).

- In cases such as for NPB-MG and NPB-CG, whose FOCAML-to-Java-compiled versions outperform their Java versions for larger values of k, those

—same as in Figure 7.11— —same as in Figure 7.11—



—same as in Figure 7.11— —same as in Figure 7.11—

OddFibonacci Chess

Figure 8.10: Speedup (relative to compiler-generated code in Chapter 6) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

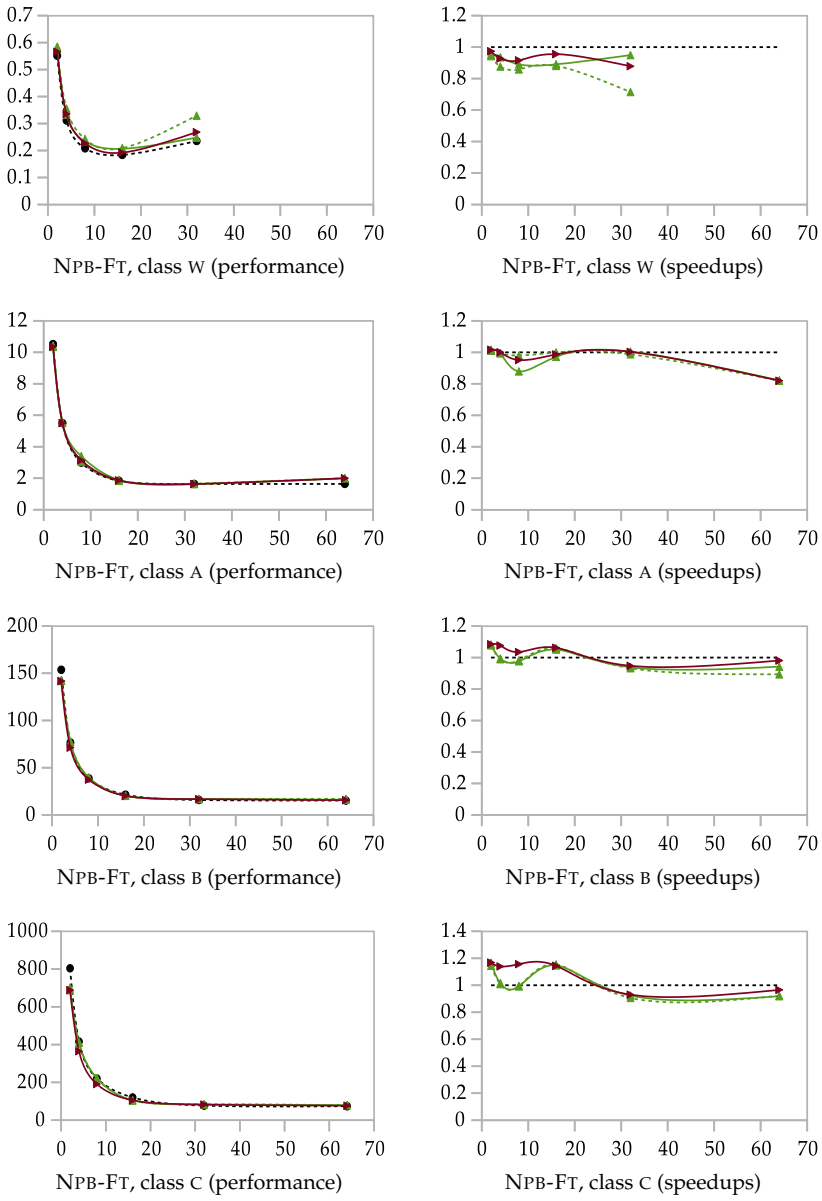


Figure 8.11: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

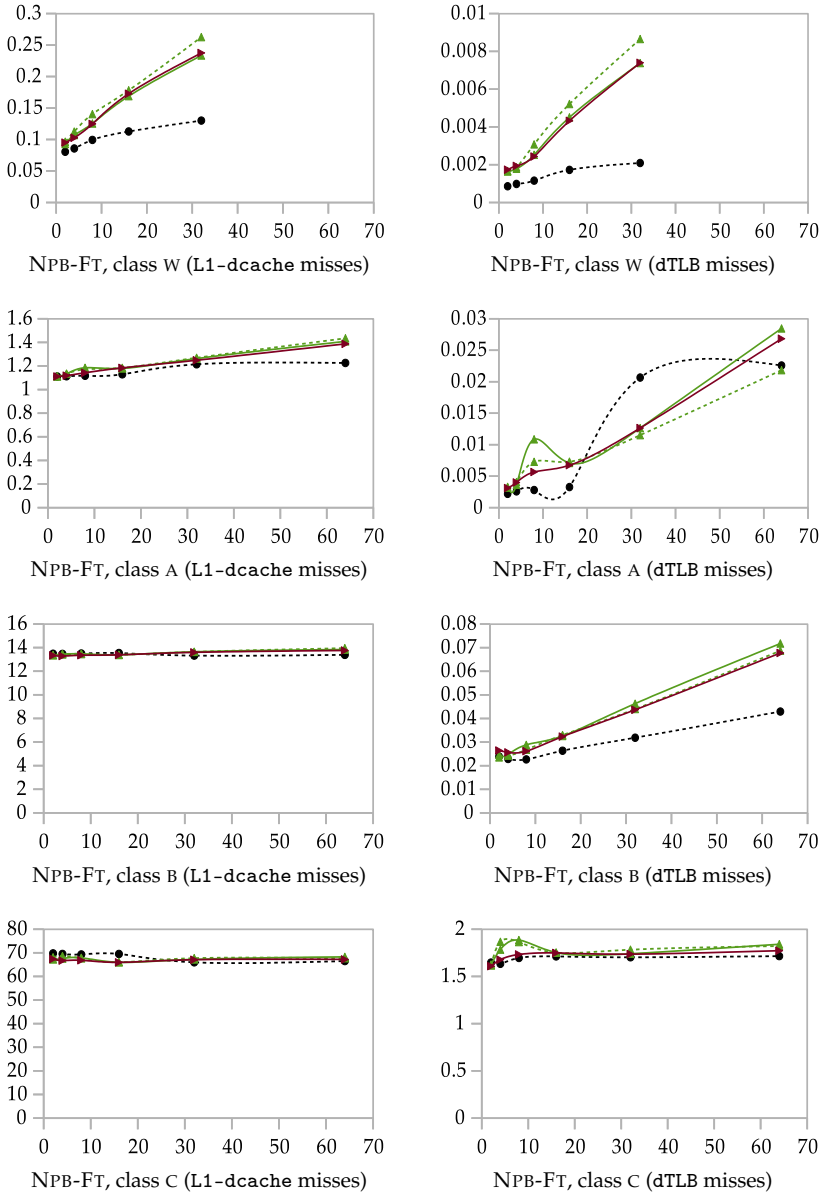


Figure 8.12: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

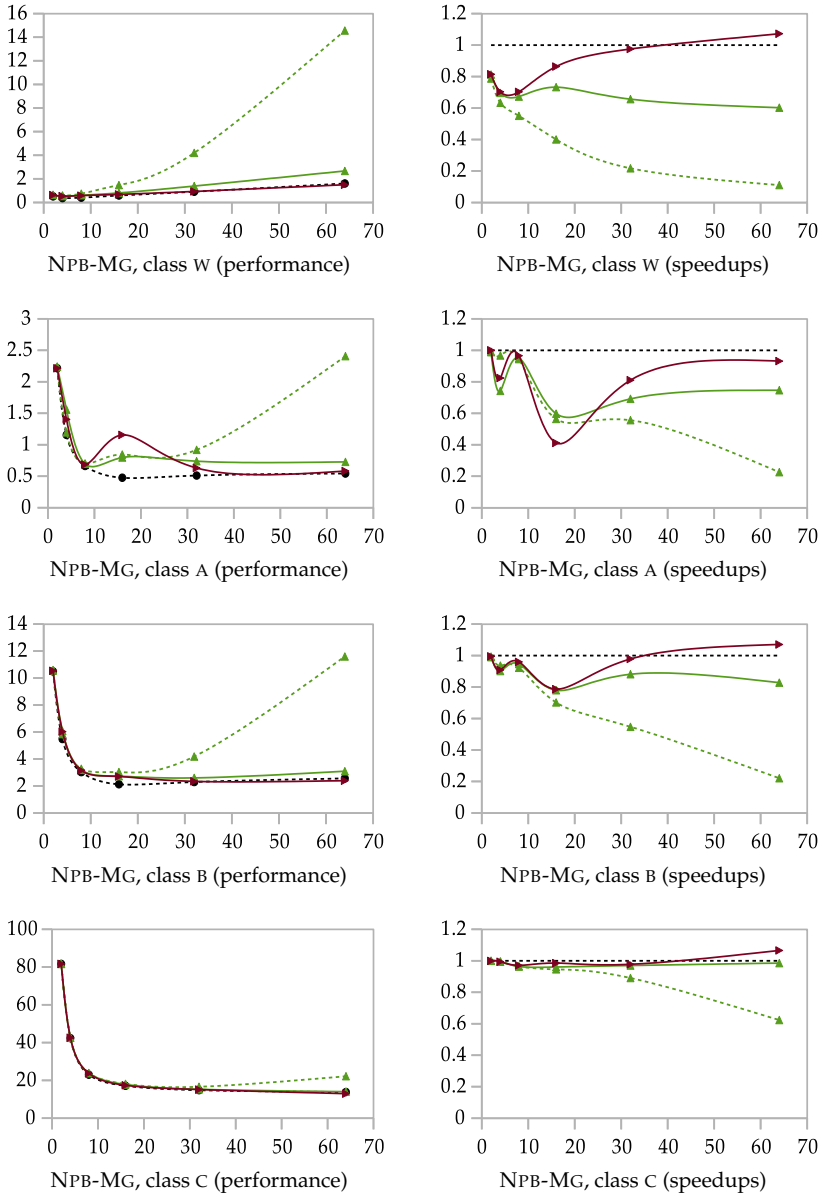


Figure 8.13: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

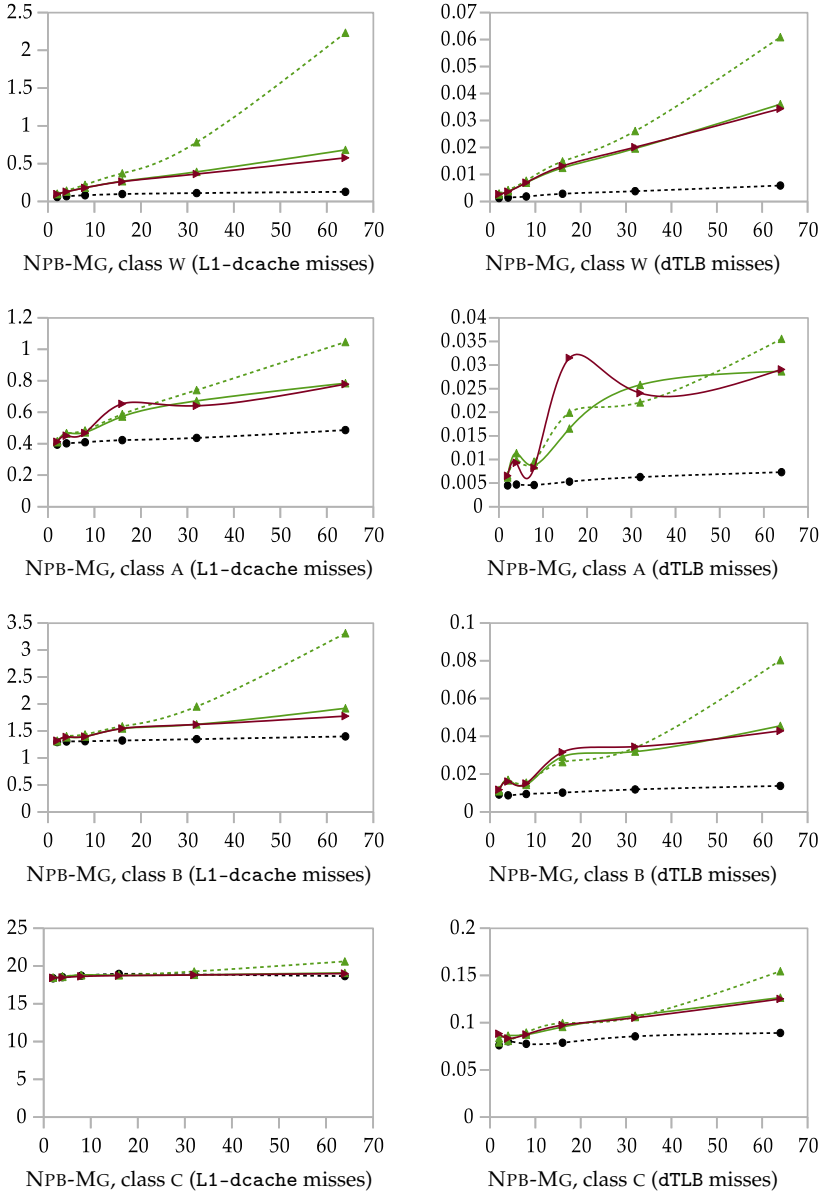


Figure 8.14: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

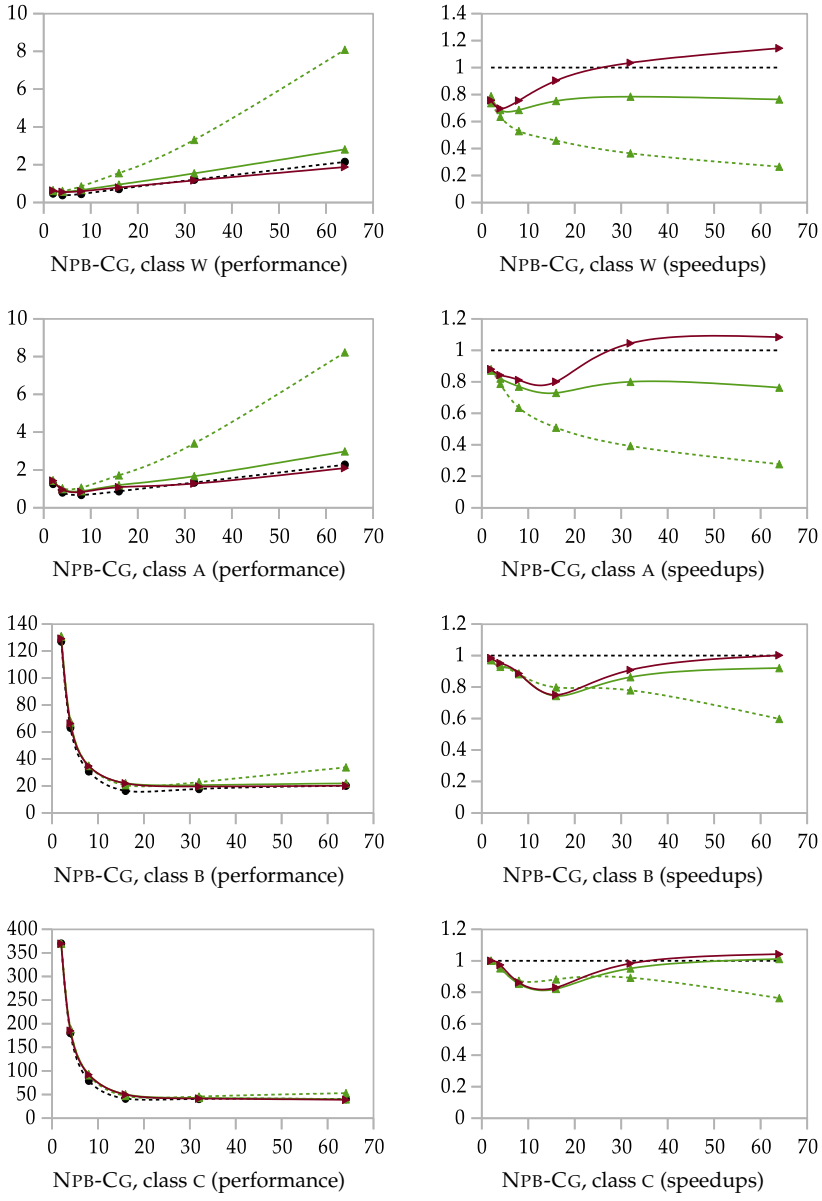


Figure 8.15: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

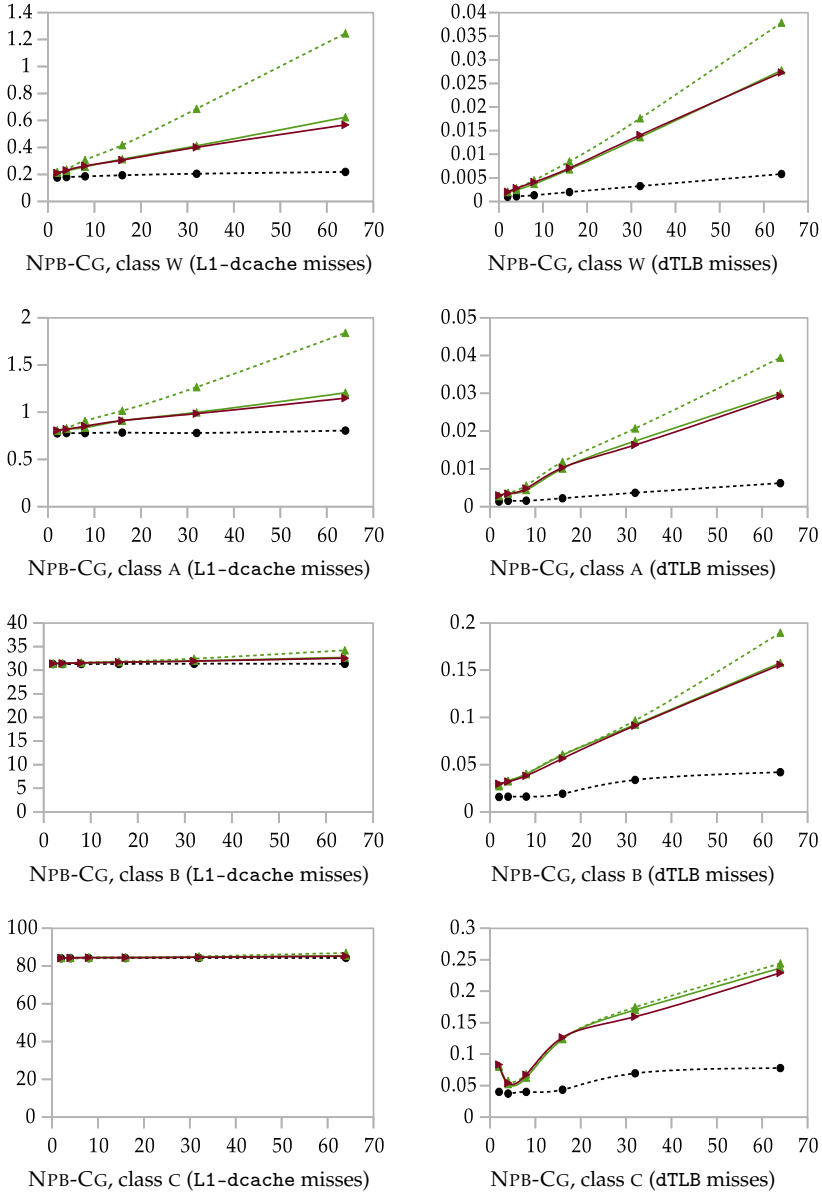


Figure 8.16: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

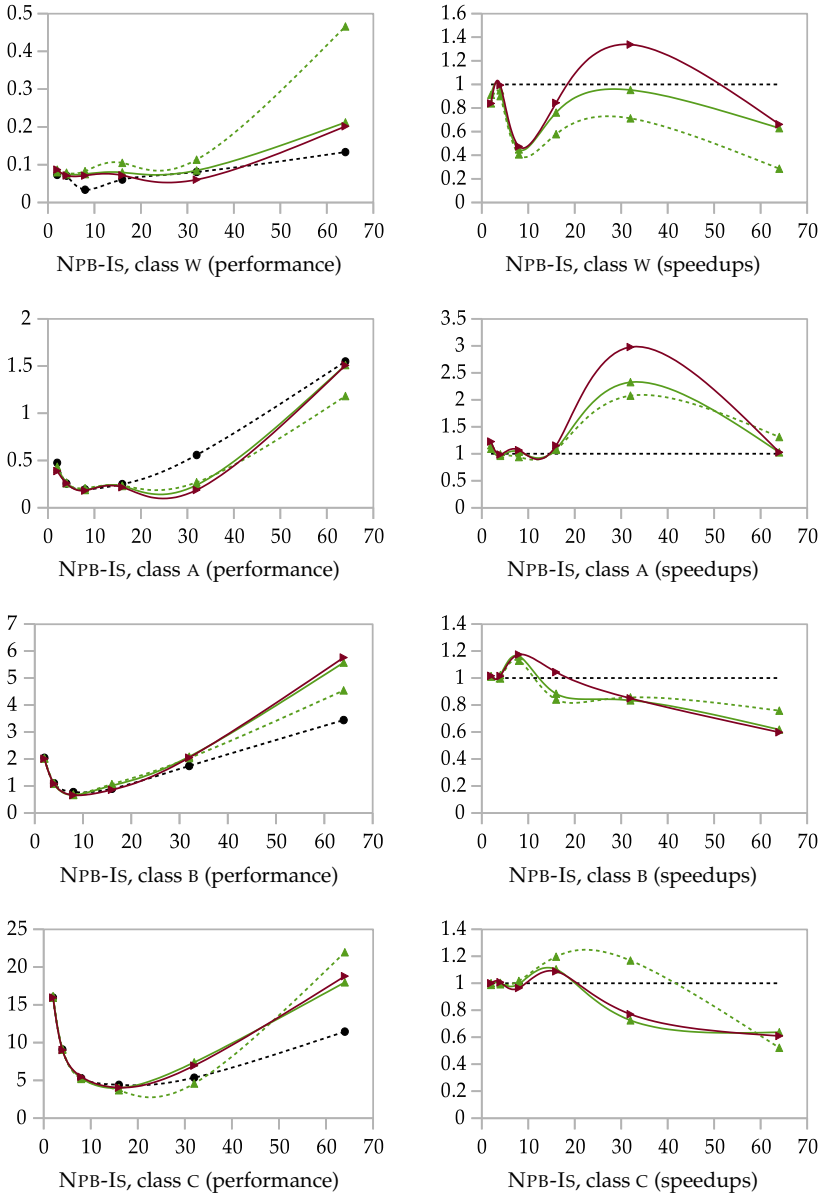


Figure 8.17: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

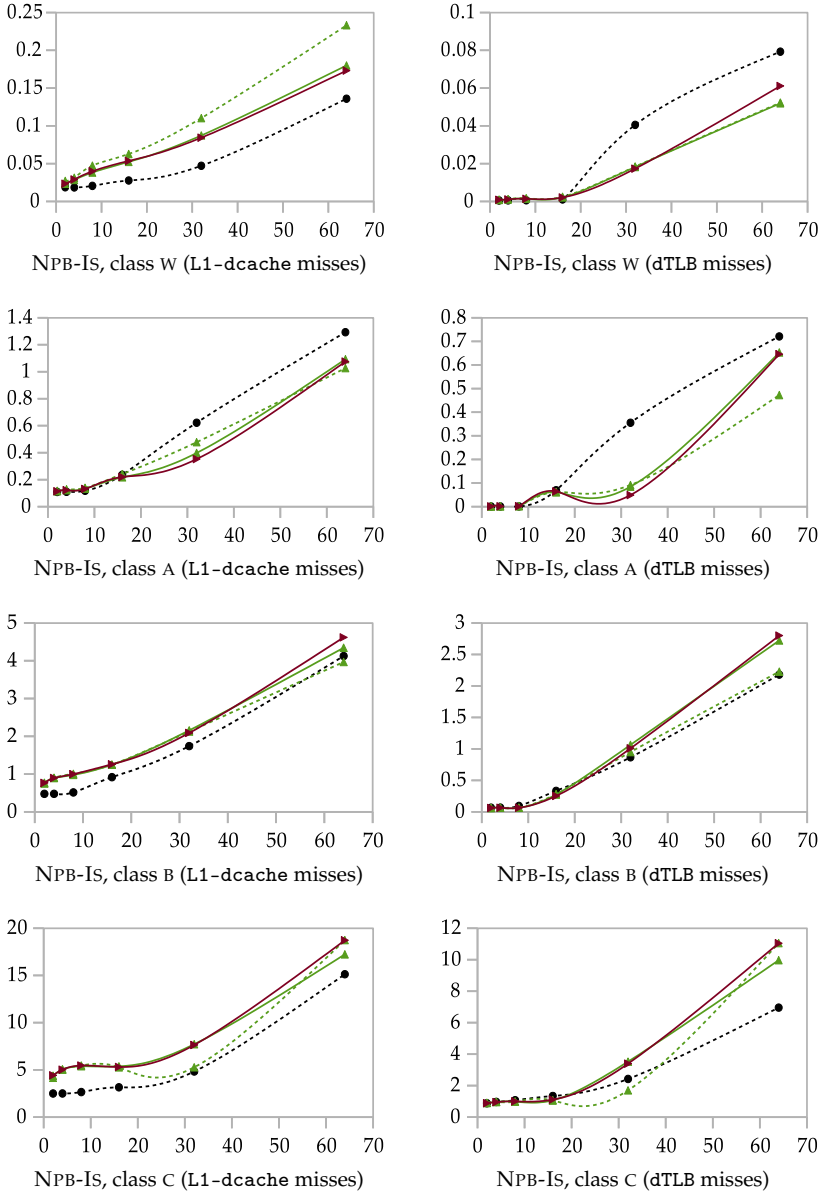


Figure 8.18: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

former versions actually incur substantially more cache misses. Assuming that these cache misses occur uniformly over the entire run time (recall from Chapter 5 that I had no choice but to measure cache misses from start to end instead of only during the interval of my time measurements, which make the numbers of cache misses reported here only an approximation), this suggests that the Java virtual machine needs to execute substantially fewer instructions for those FOCAML-to-Java-compiled versions than for those Java versions.

- An indication of improved performance and scalability, the FOCAML-to-Java-compiled versions of NPB-MG and NPB-CG outperform their Java versions not only in the larger problem size classes but also in the smaller problem size class W (improved performance), for larger values of k (improved scalability).

- As in Chapters 5, 6, and 7, differences in numbers of cache misses explain the perhaps confusing results for NPB-IS.

Figures 8.19–8.24 show performance charts for the FOCAML-to-Java-compiled versions of the NPB application benchmarks (averaged over five runs), speedup charts (with respect to their Java versions by Frumkin et al.), and charts about cache misses. The lines have the same meaning as in the figures with experimental results for the NPB kernel benchmarks. Recall from Figure 5.12 that NPB-BT and NPB-LU do not support more than 22 and 31 slaves, for which reason I have no measurements beyond $k = 16$ in class W for those benchmarks. In the same figure, note that NPB-BT, NPB-SP, and NPB-LU support at most 62 workers in class A. For that reason, as in Chapter 7, I compiled the FOCAML versions of those benchmarks for $k = 62$ instead of $k = 64$. Essentially, the same observations apply here as for the previous experimental results of the NPB kernel benchmarks.

This concludes the NPB experiments that I report on in this thesis. To summarize my main findings, the series of experiments in Chapters 5, 6, 7, and 8 show that: (i) without any improvements, the FOCAML-to-Java-compiled versions perform substantially worse than the Java versions by Frumkin et al., (ii) the performance of these FOCAML-to-Java-compiled versions improves with every new improvement I introduce, (iii) with all these improvements in place, FOCAML-to-Java-compiled versions perform roughly as well—sometimes slightly worse, sometimes slightly better—than the Java versions by Frumkin et al., and (iv) memory and cache usage comprises an important future point of attention, as I state also in Chapter 9.

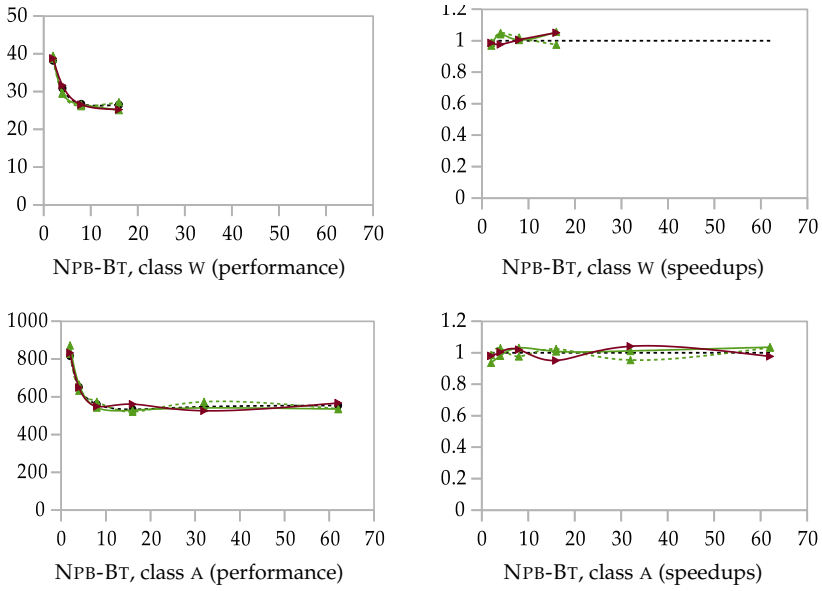


Figure 8.19: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

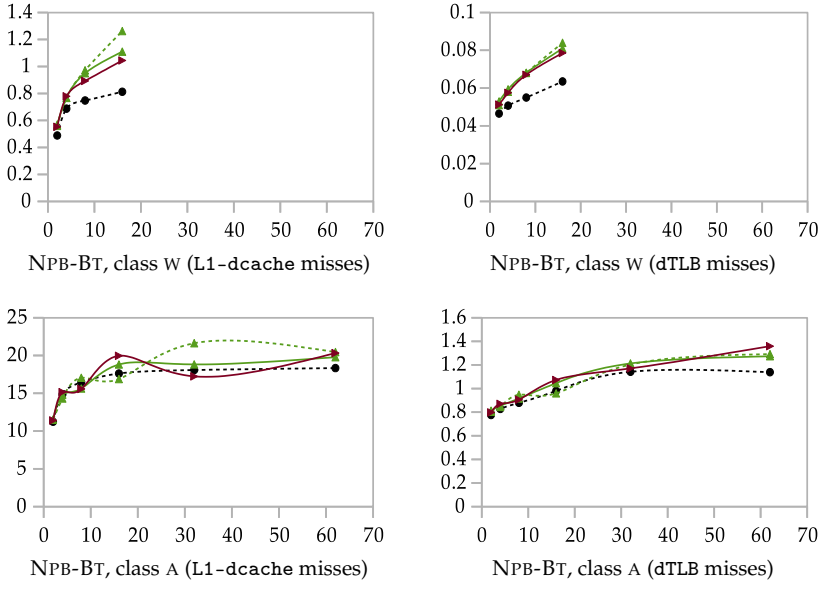


Figure 8.20: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

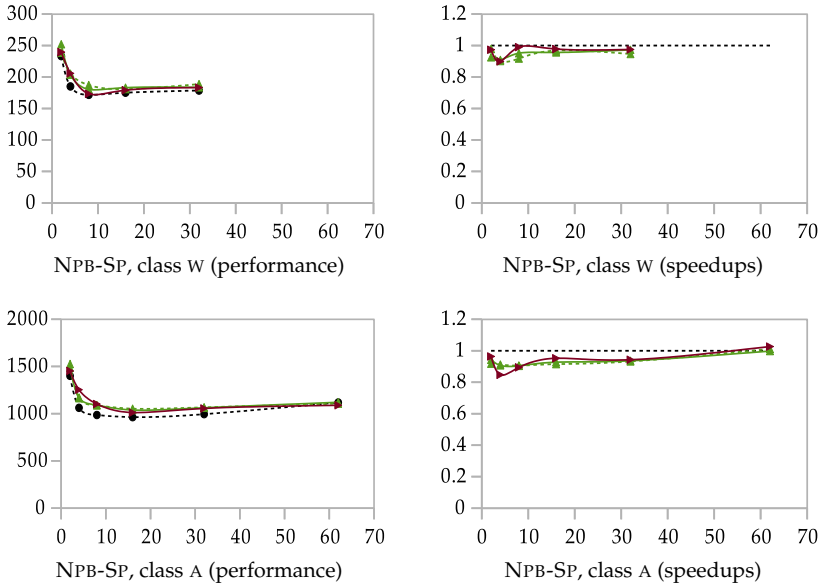


Figure 8.21: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

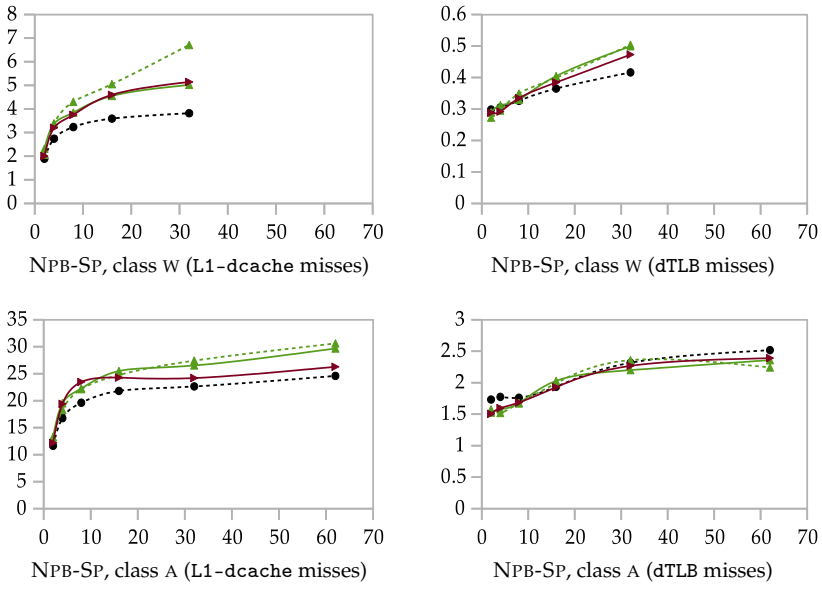


Figure 8.22: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

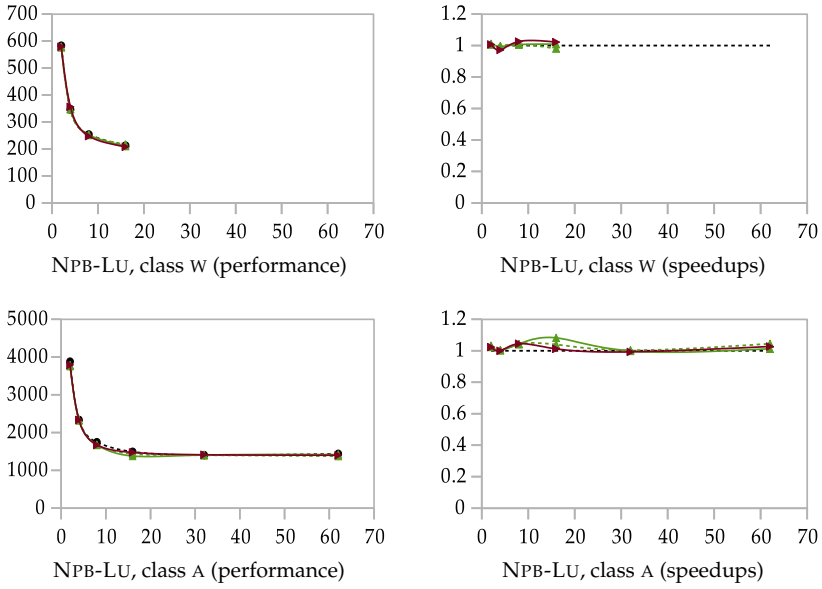


Figure 8.23: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

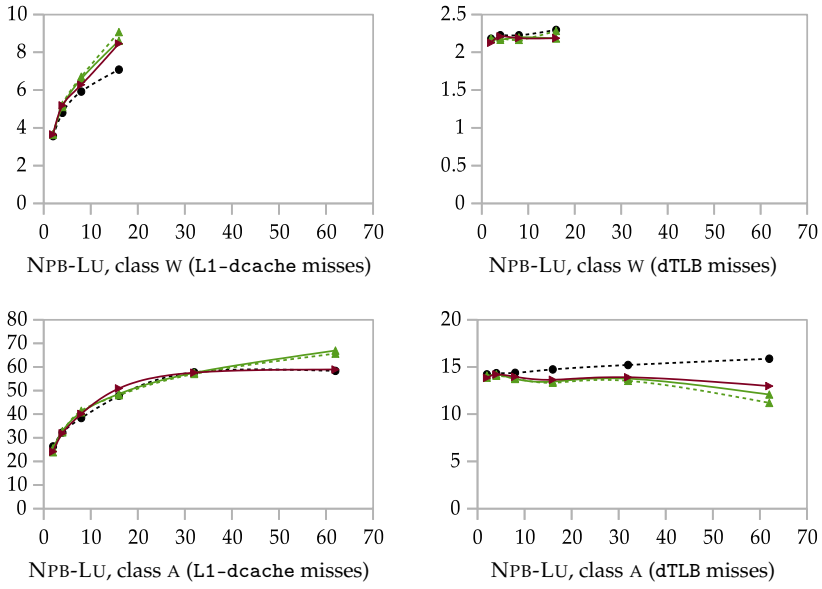


Figure 8.24: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

Chapter 9

Conclusion

9.1 Summary

In the early 2000s, hardware manufacturers ran into a number of obstacles that prevented them from directing the still exponential increase in transistors toward faster *unicore processors* [ABC+06]. In 2005, this trend led to the introduction of *multicore processors*, capable of processing multiple instruction streams in parallel. As a consequence, software engineers needed to start writing parallel programs. For optimal performance, regardless of how many cores a processor consists of—24 today or 1024 tomorrow—the same parallel program should divide instructions as evenly as possible over all its available cores. Conceptually, every parallel program consists of *workers*, which perform the actual computation, and *protocols*, which state the rules of interaction that workers must abide by. To achieve good performance on multicore processors, by *Amdahl's Law* [Amd67], software engineers should minimize the inherently sequential fraction of computation performed by workers. Additionally, and of increasing importance as this sequential fraction decreases [YMG14], software engineers should minimize the amount of resources spent on enforcing protocols. [Section 1.1]

A decade after the advent of multicore processors, many software engineers still use largely the same abstractions for parallel programming in general, and implementing protocols in particular, as forty years ago: threads, shared memory, and concurrency constructs for mutual exclusion [Dij02, Hoa74]. Even when programming with higher-level abstractions, such as *thread pools* [DM98, GPB+06, LSB09, Rei07, Rob13] or *actors* [AVWW96, Hal12, HO09], the implementation of nontrivial protocols often requires software engineers to use these relatively old techniques [TDJ13]. However, implementing protocols by manually managing shared memory with concurrency constructs for mutual exclusion has three major issues: (i) it complicates correctly implementing protocols, because it complicates reasoning about programs' behavior, (ii) it complicates implementing protocols with high performance, because it fails to preserve

crucial *intention information* that compilers need for automatically optimizing interaction code, and (iii) it complicates implementing protocols in a modular fashion, because it neither enforces nor encourages syntactic separation of computation code from interaction code. One way to resolve these issues consists of imposing another level of abstraction on top of shared memory, thereby eliminating the need for concurrency constructs for mutual exclusion for implementing protocols. Incarnations of this solution include *transactional memory* [HM93, Kni86, ST97] and *algorithmic skeletons* [Col88, GL10], but both these solutions have their limitations. [Section 1.2]

In another incarnation of this solution, software engineers implement their protocols at a new, high, *intention-expressing level of abstraction*, which provides constructs for implementing protocols while preserving as much information about software engineers' intention behind those implementations as possible. Not only should this make correctly implementing protocols easier for software engineers (thereby resolving the first issue, above), but it also makes automatically optimizing their resulting protocol implementations easier for compilers (thereby resolving the second issue). *Domain-specific languages* (DSL) for interaction seem a particularly well-suited vehicle for providing software engineers such intention-expressing levels of abstraction, through intuitive and protocol-tailored syntax. DSLs for interaction also force software engineers to syntactically separate protocol implementations from worker implementations (thereby resolving the third issue), and they naturally complement existing *general-purpose languages* (GPL): software engineers can continue implementing their workers in a GPL, but their protocols in a DSL, after which a DSL compiler translates those DSL-coded protocol implementations into GPL code. Together, hand-written GPL-coded worker implementations and compiler-generated GPL-coded protocol implementations constitute full GPL-coded parallel programs, whose grand integration can happen automatically, by the DSL compiler, completely transparent to software engineers. [Section 1.3]

In the programming model for one concrete instance of the previous approach, every worker has access to a number of conceptual *ports*. Through such ports, workers can interact with their "environment", by performing blocking I/O operations on those ports. Crucially, however, a worker never knows which other workers constitute its environment—every worker sees only its own ports and cannot directly address other workers or their ports. Instead, whenever a worker performs an I/O operation, a separate entity responsible for enforcing the protocols among workers determines whether this I/O operation may immediately complete—perhaps synchronously with already pending I/O operations performed by other workers—and if so, how data flow between ports; if not, the worker becomes *suspended* and remains suspended until its I/O operation completes at some future time. A true intention-expressing DSL for interaction, then, must provide constructs for expressing "a set of rules that control the way data is [exchanged through ports]", which, in fact, constitutes a dictionary definition of "protocol" (i.e., a generally accepted interpretation, and closest approximation of, the intention that people have when they use the word "protocol"). [Section 1.3]

Every set of atomic data-flows between ports yields one *instance of interaction* among workers. A simple formal model of instances of interaction, then, consists of functions from ports (involved in an instance of interaction) to data (observable on those ports in that instance). An infinite sequence of such functions subsequently models one *chain of interaction* admitted by a protocol; a set of such infinite sequences models all that protocol's admissible infinite chains. Interpreting the latter kind of sets (and their elements) as automata-theoretic *languages* (and infinite *words*), a concise formal model of protocols consists of automata over such languages. Essentially, every transition of such an automaton represents one rule "that control[s] the way data is [exchanged through ports]", and as such, this kind of automata truly captures the intention behind the word "protocol". [Section 2.1]

In one rather naive automaton model of protocols, transitions explicitly carry functions from ports to data as their labels. Often, however, this gives rise to infinitely many transitions. Therefore, in a more advanced automaton model of protocols—*constraint automata*—transition labels symbolically represent (possibly infinite) sets of functions from ports to data as two constraints: a *synchronization constraint*, which consists of a set of ports, and a *data constraint*, which consists of a first-order logic formula. Conceptually, a synchronization constraint specifies which ports synchronize in an admissible instance of interaction (i.e., through which ports data synchronously flow); a data constraint specifies the particular data observable on those ports. By definition, *behaviorally equivalent* constraint automata accept the same language (i.e., their corresponding protocols admit exactly the same chains of interaction). Analogously, *behaviorally congruent* constraint automata also accept the same language, but moreover, they also have coincident transition relations (in a *bisimulation* kind of way [BSAR06, Mil89]). Although only behavioral equivalence matters in the end, behavioral congruence often simplifies proofs and reasoning about constraint automata. While individual constraint automata model individual protocols, their *multiplication* and *subtraction* model protocol composition (i.e., superimposing different sets "of rules that control the way data is [exchanged through ports]") and abstraction (i.e., removing ports not of interest). [Section 2.1]

A Java library for constraint automata allows software engineers to represent constraint automata, their multiplication, and their subtraction in Java, as nonexecutable data structures. [Section 2.2]

Because constraint automata model (the intention behind) protocols, they constitute a well-suited semantic domain for an intention-expressing DSL for interaction. As in any kind of engineering, rather than providing software engineers syntactic constructs for directly constructing constraint automata, a more scalable approach consists of providing software engineers constructs for representing *multiplication expressions* of constraint automata, thereby exploiting their compositionality. In a graphical syntax for this approach, software engineers draw multiplication expressions as data-flow diagrams. This graphical syntax essentially yields *Reo* [Arb04, Arb11], an existing graphical language

for compositional construction of protocols. In an alternative textual syntax, called FOCAML, software engineers write multiplication expressions as declarative pieces of code. Reo and FOCAML have complementary use cases: Reo visualizes data-flows, which makes seeing—at a glance—which rules of interaction a protocol enforces easier, while FOCAML has more expressive power in terms of parametrization. [Section 3.1]

The FOCAML editor/parser/interpreter plugin for Eclipse 4.x enables software engineers to write FOCAML programs. This plugin can also translate Reo diagrams, drawn using an existing collection of plugins for Reo development, into FOCAML code. [Section 3.2]

To compile FOCAML code (and, using the Reo-to-FOCAML translator, also Reo diagrams) into GPL code to get a full GPL-coded program, several approaches exist. In the *Distributed Approach*, a FOCAML compiler translates every primitive constraint automaton in a multiplication expression into its own thread. These threads use an expensive consensus algorithm to compose their local behavior into consistent global behavior, effectively computing their product at run-time. In the *Centralized Approach*, in contrast, a FOCAML compiler first computes the product of the primitive constraint automata in a multiplication expression, then subtracts all the resulting *internal ports* (i.e., ports that serve as an *input port* in one constraint automaton and as an *output port* in another one, which nobody can access anymore in their product), and finally translates the resulting composite constraint automaton into a single thread. [Section 4.1]

GPL code generated from FOCAML code under the Distributed Approach exhibits maximal parallelism (with respect to the primitive constraint automata in a multiplication expression) and, therefore, achieves relatively high *throughput*. The expensive *consensus algorithm* required in the Distributed Approach, however, inflicts serious overhead. Code generated under the Distributed Approach, therefore, suffers from relatively high *latency*. In contrast, code generated under the Centralized Approach exhibits maximal sequentiality and, therefore, suffers from relatively low throughput. By avoiding the need for expensive consensus as in the Distributed Approach, however, the Centralized Approach eliminates a source of serious overhead. Code generated under the Centralized Approach, therefore, achieves relatively low latency. [Section 4.1]

Without additional compiler flags, the FOCAML-to-Java compiler plugin for Eclipse 4.x compiles FOCAML code into Java code under the Centralized Approach. Built on top of the FOCAML editor/parser/interpreter plugin, it also supports compiling Reo into Java. [Section 4.2]

The Centralized Approach has two scalability problems, one of which manifests at compile-time, the other of which manifests at run-time. At compile-time, computing the product of all primitive constraint automata in a multiplication expression may give rise to *state space explosion*; at run-time, the single thread for a computed product may give rise to *oversequentialization*. In oversequentialized compiler-generated code for a constraint automaton, two *independent transitions*, originating from two independent primitive constraint

automata in a multiplication expression (i.e., constraint automata that share no ports), cannot fire simultaneously but only consecutively, thereby unjustifiably reducing throughput. To solve these two problems, while avoiding the need for expensive consensus as in the Distributed Approach, a FOCAML compiler can apply the *Hybrid Approach*. This third approach sits somewhere between the Distributed Approach and the Centralized Approach, by serializing *useless parallelism* (i.e., the kind of parallelism that requires heavy synchronization) to preserve only *useful parallelism* (i.e., the kind of parallelism that requires light synchronization). [Section 5.1]

In the Hybrid Approach, a FOCAML compiler first distributes the primitive constraint automata in a multiplication expression over disjoint subsets. Subsequently, as in the Centralized Approach, the compiler computes *per-subset products* by multiplying, for every subset, the constraint automata in that subset (and by subsequently subtracting internal ports from those per-subset products). Finally, as in the Distributed Approach, the compiler translates the resulting per-subset composite constraint automata into as many threads. Contrasting the Distributed Approach, however, these threads require only a cheap consensus algorithm for synchronizing their behavior. [Section 5.1]

Just as the expensive consensus algorithm required in the Distributed Approach, the cheap consensus algorithm required in the Hybrid Approach corresponds to run-time multiplication of constraint automata, but under a new definition of multiplication. This new multiplication, always called *l(ocal)-multiplication*, generally does not coincide with the old multiplication, sometimes called *g(lobal)-multiplication*. For instance, l-multiplication does not exhibit associativity, whereas g-multiplication does. Only g-multiplication matters in the end: a FOCAML compiler always starts with a g-multiplication expression, which forms an absolute reference point for compilation soundness. In the worst case, then, the threads generated in the Hybrid Approach do not behave as their g-multiplication expression (because they use the cheap consensus algorithm, which corresponds to l-multiplication), making their behavior unsound. However, by carefully *partitioning* the primitive constraint automata in a g-multiplication expression into subsets (i.e., the first step in the Hybrid Approach), a FOCAML compiler can guarantee that the behavior of their corresponding threads corresponds to g-multiplication, even though those threads use the cheap consensus algorithm. In Reo terminology, such partitioning corresponds to computing *synchronous and asynchronous regions* [CP12, **J**CP12, **J**CP16, PCdVA11, PCdVA12, Pro11]. [Section 5.1]

With compiler flag `PARTITION` raised, the FOCAML-to-Java compiler plugin for Eclipse 4.x compiles FOCAML code into Java code under the Hybrid Approach. [Section 5.2]

One problem that the Hybrid Approach, as the Centralized approach, still suffers from concerns the size of data constraints, symptomized by the fact that the neutral element for multiplication negatively affects performance of compiler-generated code: multiplying this neutral element any number of times with other constraint automata behaviorally makes no difference, but performance-

wise, it does. This problem results from the definition of subtraction, which a FOCAML compiler uses for removing internal ports (from per-subset products in the Hybrid Approach or from the full product in the Centralized Approach). In particular, the original definition of subtraction removes ports from data constraints in constraint automata only *semantically*, by enveloping each of its data constraints in an existential quantification for the port-to-remove. By doing so, subtracting a port from a constraint automaton actually increases the sizes of its data constraints. Larger data constraints generally require more computational resources to handle at run-time, which requires constraint solving over a finite discrete domain. This, then, explains why the neutral element for multiplication negatively affects performance: it yields equivalent, yet substantially larger, data constraints. [Section 6.1]

Syntactic subtraction, a new subtraction on constraint automata, avoids this problem. Instead of enveloping data constraints in existential quantifications, it tries to find a suitable—semantically neutral—substitute for the port-to-remove in every data constraint. If such substitutes exist, syntactic subtraction subsequently replaces every occurrence of the port-to-remove in a data constraint with its substitute in that data constraint. By subsequently removing obvious tautologies, syntactic subtraction actually decreases data constraint sizes. The previous problem with the neutral element for multiplication then also goes away. [Section 6.1]

Syntactic subtraction works effectively only when applied to *normalized constraint automata*. Normalization, applicable to every well-formed constraint automaton, therefore constitutes an important operation. [Section 6.1]

With compiler flag `SUBTRACT_SYNTACTICALLY` raised, the FOCAML-to-Java compiler plugin for Eclipse 4.x compiles FOCAML code into Java code using syntactic subtraction (instead of semantic subtraction). [Section 6.2]

Notwithstanding syntactic subtraction, handling data constraints with general-purpose *constraint solving* techniques inflicts a significant amount of overhead at run-time—not only overhead proportional to the size of a data constraint but also a constant overhead for preparing, making, and processing the result of every constraint solver invocation—especially for relatively simple data constraints. Many such data constraints, however, essentially constitute declarative specifications of data-flows between ports. When provided such specifications, then, most—if not all–software engineers would probably just write direct imperative implementations instead of indirect invocations to a constraint solver. [Section 7.1]

Commandification, an operation on constraint automata, aims to automate just that: it translates every data constraint in a constraint automaton into a piece of imperative code, called a *data command*, in a behavior-preserving way (proved by equipping data commands with a transition system semantics, notions of partial/total correctness, and Hoare logic [AdBO09, Hoa69]). The construction of data commands for data constraints uses hypergraph representations of the data-flows represented by those data constraints, paying special attention to cycles in such hypergraphs. [Section 7.1]

With compiler flag `COMMANDIFY` raised, the FOCAML-to-Java compiler plugin for Eclipse 4.x compiles FOCAML code into Java code using commandification. [Section 7.2]

To fire any transition in a constraint automaton at run-time, its corresponding thread first needs to check all transitions out of the current state for enabledness (regardless of whether this thread came about through the Distributed, Centralized, or Hybrid Approach). This requires $\mathcal{O}(k)$ time, where k denotes the number of transitions. Often, the number of transitions increases at least linearly as the number of workers increases. Consequently, often, the time required to check transitions for enabledness also increases at least linearly as the number of workers increases. This may cause the performance of compiler-generated code to scale poorly. [Section 8.1]

Queue-inference aims to solve this problem in cases where the transitions to check for enabledness differ from each other in a "well-behaved way". For instance, suppose that all k transitions involve only two ports: the same output port but a different input port in each transition. Originally, to check these transitions for enabledness, a thread checks both of the two ports involved in each of these transition for a pending I/O operation. But now, suppose that whenever a worker performs an I/O operation on a port, it also offers that port into a special *queue*. To *simultaneously* check all k transitions for enabledness, then, a thread needs to check only this queue for nonemptiness (in addition to checking the output port for a pending I/O operation); if so, this thread knows that at least one of the k transitions can fire. Consequently, k checks for enabledness collapse into just one. Automata-theoretically, such collapsing corresponds to *combining* multiple transitions into a single transition with a special transition label, to express that this combined transition has an efficient queue-based implementation. With queue-inference, then, a FOCAML compiler first analyzes constraint automata for transitions amenable to combination, subsequently actually combines such transitions, and finally generates code with queues for combined transitions. [Section 8.1]

With compiler flag `INFER_QUEUES` raised, the FOCAML-to-Java compiler plugin for Eclipse 4.x compiles FOCAML code into Java code using queue-inference. [Section 8.2]

Figures 9.2 and 9.3 show experimental results (absolute performance and relative speedups) for eight *families* of constraint automata, each parametric in a natural number k. Blue lines represent code generated under the Centralized Approach without further improvements; red lines represent code generated under the Hybrid Approach; yellow lines represent code generated under the Hybrid Approach, plus syntactic subtraction; green lines represent code generated under the Hybrid Approach, plus syntactic subtraction, plus commandification; purple lines represent code generated under the Hybrid Approach, plus syntactic subtraction, plus commandification, plus queue-inference. If a chart excludes one of these lines, applying the improvement to which this missing line corresponds does not actually change the generated code (i.e., the

Centralized Approach	[Chapter 4]
Hybrid Approach	[Chapter 5]
+ syntactic subtraction	[Chapter 6]
+ commandification	[Chapter 7]
+ queue-inference	[Chapter 8]

Legend

- In performance charts, every *solid colored line* represents the performance of compiler-generated code, in millions of rounds (more = better).
- In performance charts, every *dotted colored line* represents inverse-proportional growth relative to its solid colored line at $k = 1$.
- In speedup charts in Chapters 4–8, every *solid black line* represents the speedup of compiler-generated code in the current chapter relative to the dotted black line in the same chart.
- In every speedup chart in Chapters 4–8, every *dotted black line* represents the speedup of compiler-generated code in the previous chapter relative to itself (i.e., 0% speedup, i.e., a baseline).
- In speedup charts in Chapter 9, every *solid colored line* whose color corresponds to Chapter i represents the speedup of compiler-generated code in Chapter i relative to the dotted colored line in the same chart.
- In speedup charts in Chapter 9, every *dotted colored line* whose color corresponds to Chapter i represents the speedup of compiler-generated code in Chapter i relative to itself (i.e., 0% speedup, i.e., a baseline).

Description

Figure 9.1: Legend for charts about protocol experiments

improvement has no effect). [Sections 4.2, 5.2, 6.2, and 7.2]

Figures 9.2 and 9.3 show that starting from the Centralized Approach, overall, performance gets better with every successive improvement, while software engineers no longer need to worry about manually applying these kinds of optimizations to their protocol implementations. For instance, in the Hybrid Approach, only the compiler decides about how to parallelize protocol implementations, completely transparent to software engineers. Orthogonally, syntactic subtraction automatically simplifies data-flows, essentially by eliminating intermediate vertices in a data-flow graph, while commandification linearizes declarative specifications of data-flows into efficient imperative implementations, as carefully ordered sequences of instructions. Perhaps most advanced, with queue-inference, the compiler can—all by itself—identify cases

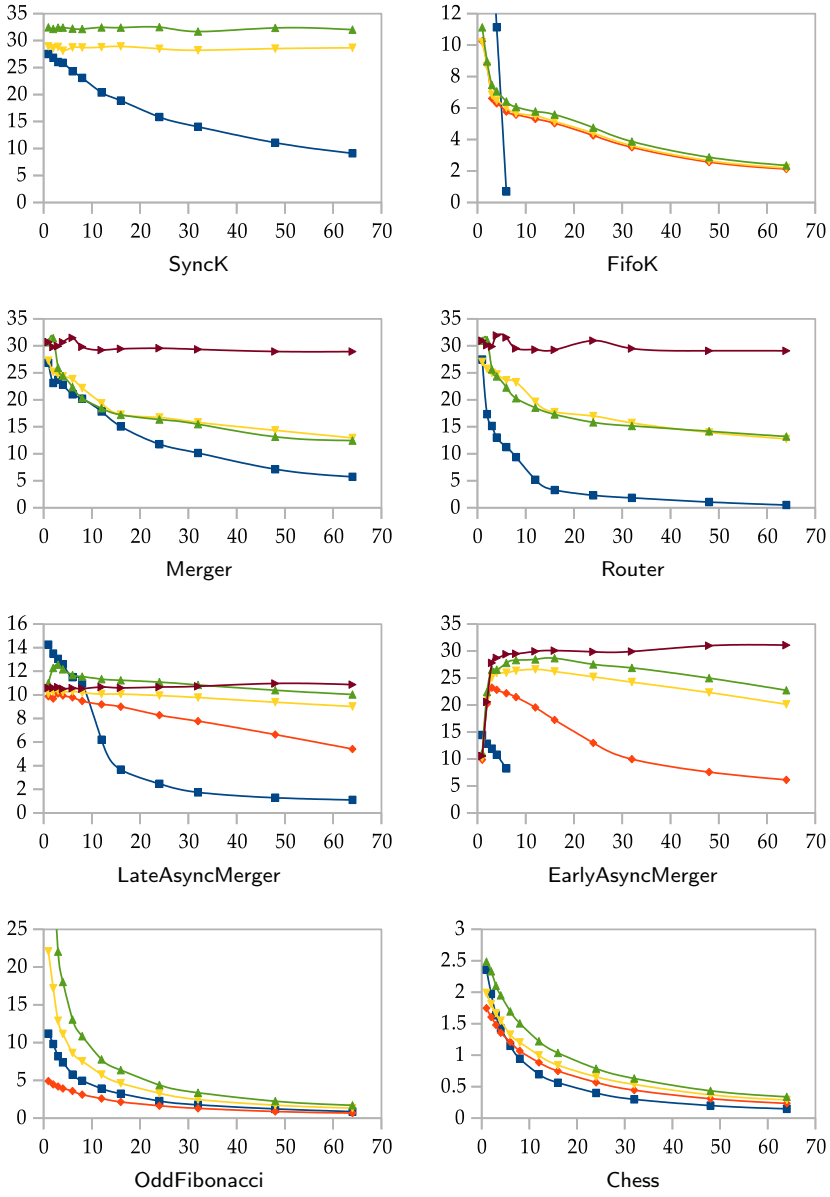


Figure 9.2: Performance (in number of completed rounds per four minutes) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See also Figure 9.1.

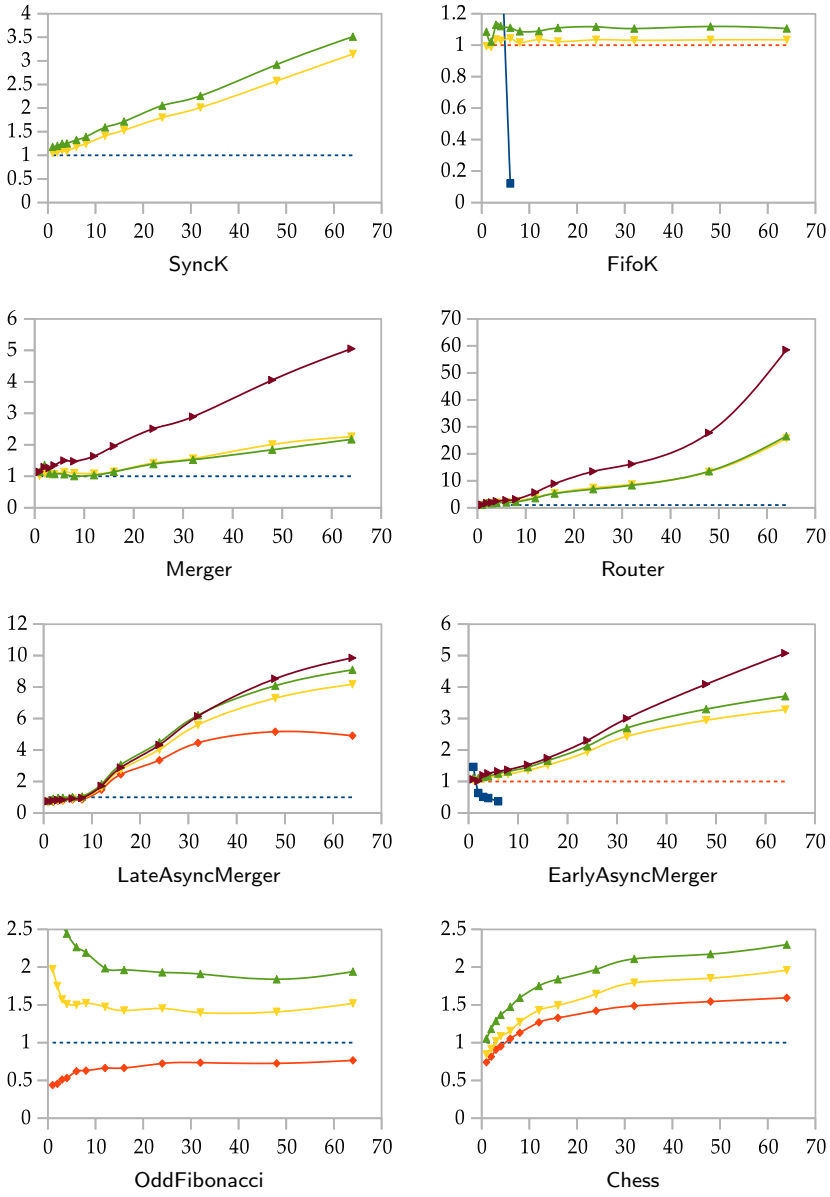


Figure 9.3: Speedup (relative to compiler-generated code in Chapter 4 and, for FifoK and EarlyAsyncMerger, relative to compiler-generated code in Chapter 5) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

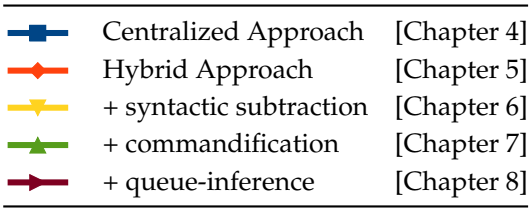


Figure 9.4: Legend for program experiment charts

where the use of queue data structures yields more scalable protocol implementations than individual variables would yield. Relieving software engineers from the responsibility of carrying out such optimizations—which, one way or the other, they themselves would have to carry out otherwise—simplifies their development of not only correct but also efficient protocol subprograms. And in each of these optimizations, the intention information captured by constraint automata plays an essential role in their automation, both at design-time (defining and modeling the optimization) and at compile-time (testing the applicability of the optimization for a given input). [Section 1.3]

Figures 9.5–9.11 show experimental results for FOCAML-to-Java-compiled versions of NASA's well-established *NAS Parallel Benchmarks* (NPB) [BBB+91, BBB+94], including experimental results for the Java versions of these bench-

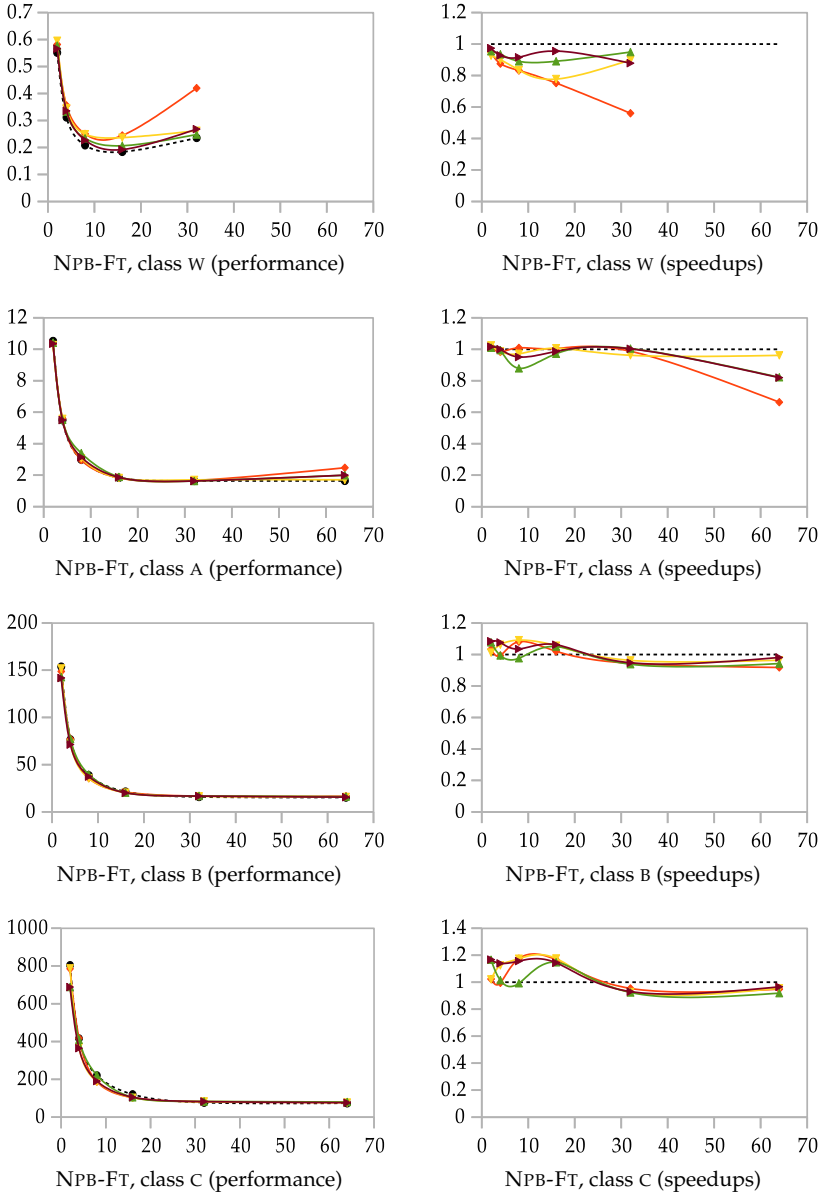


Figure 9.5: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

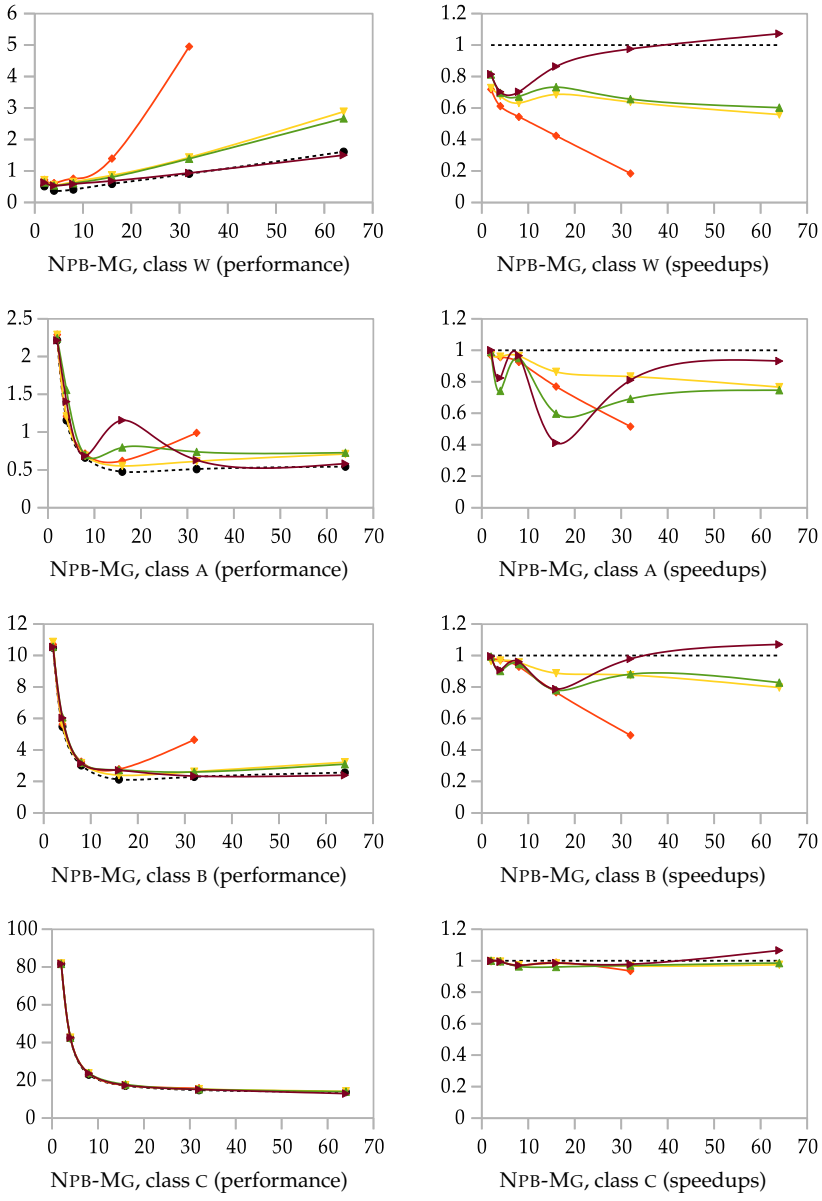


Figure 9.6: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

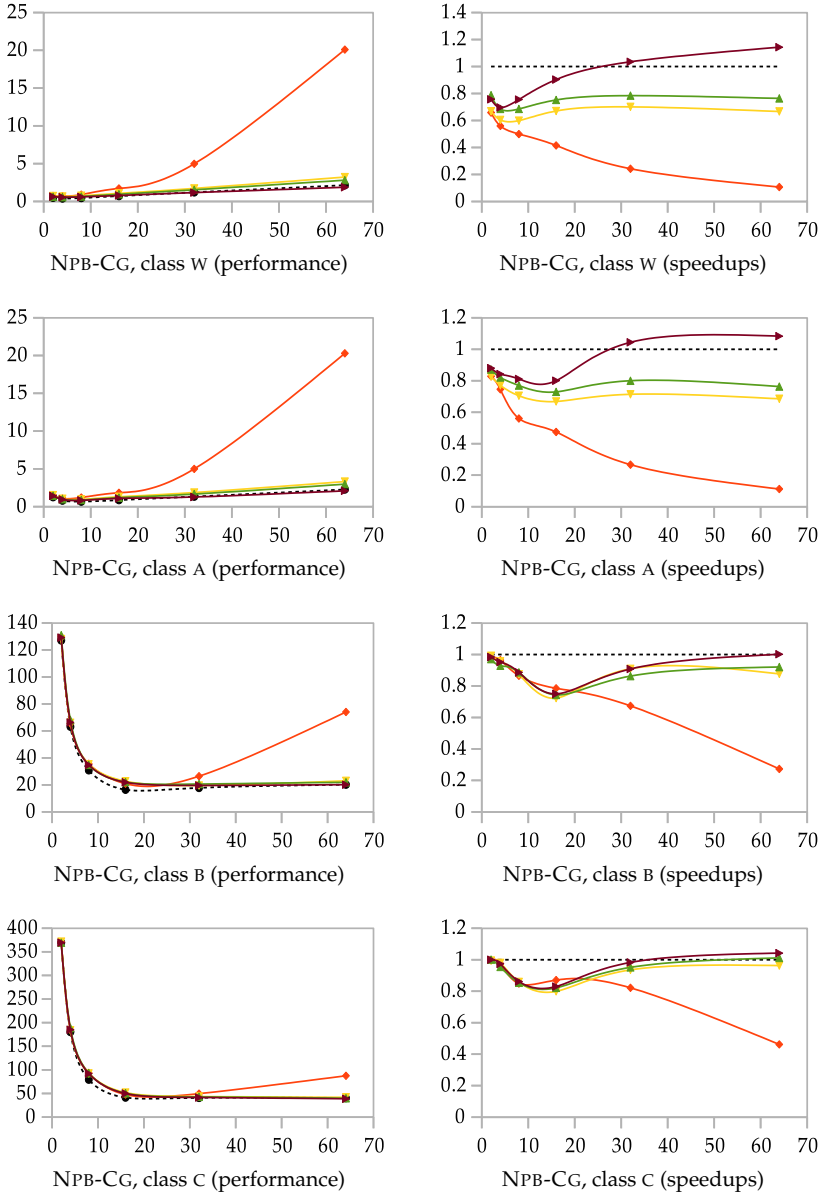


Figure 9.7: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

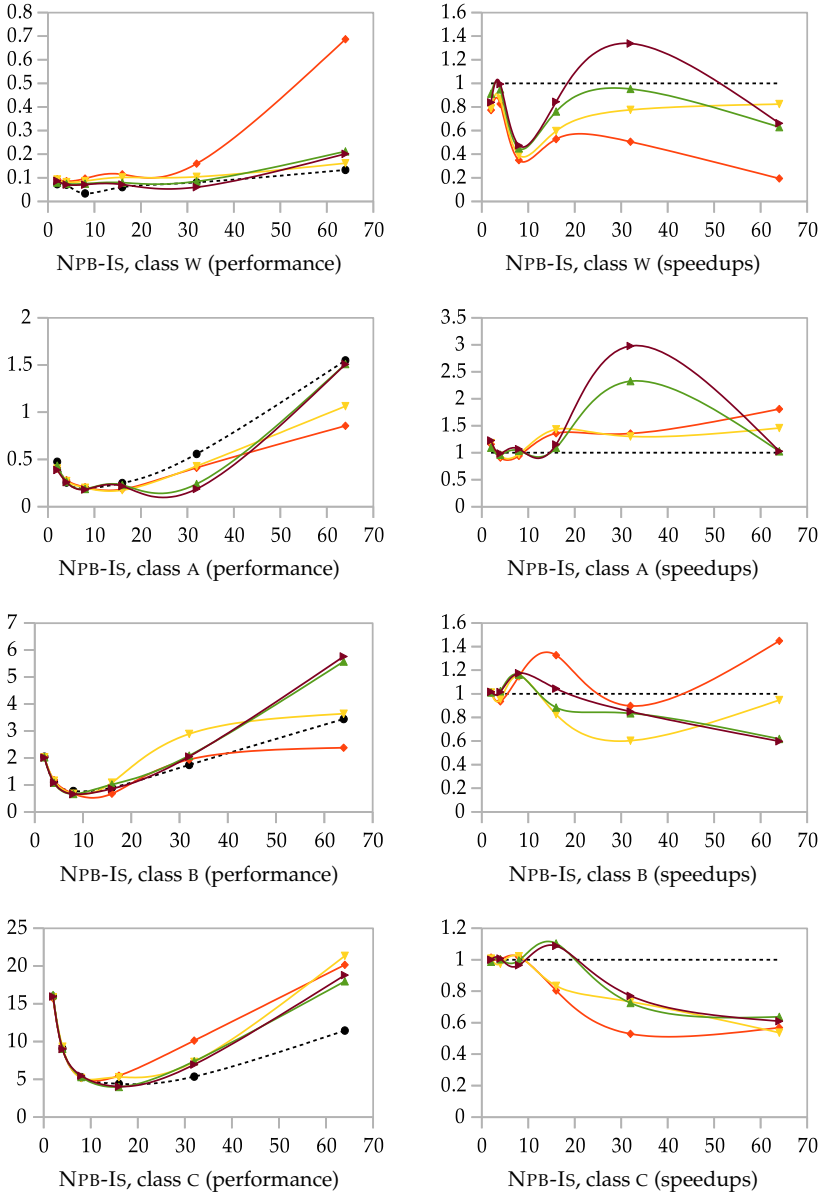


Figure 9.8: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

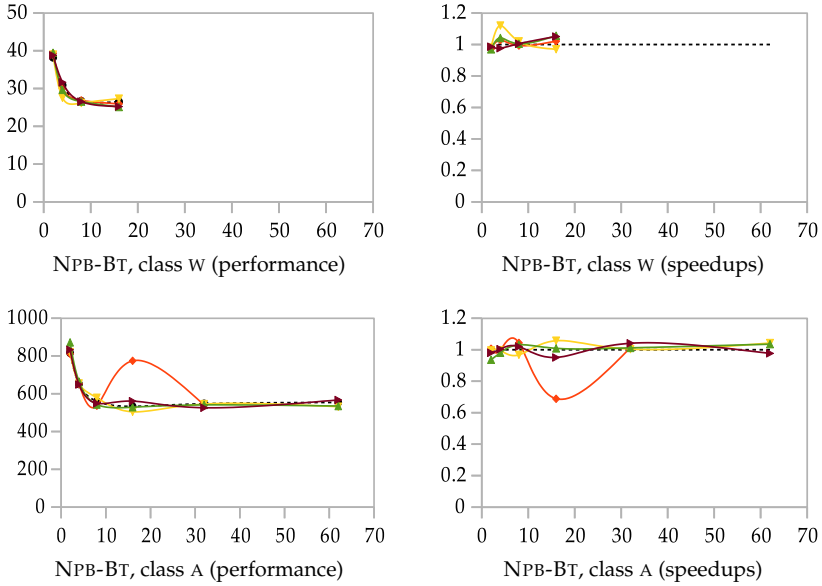


Figure 9.9: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

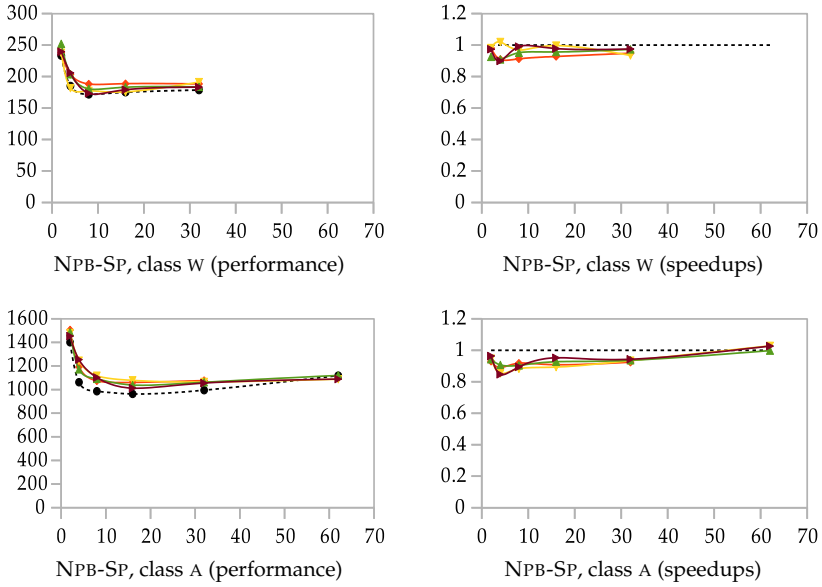


Figure 9.10: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

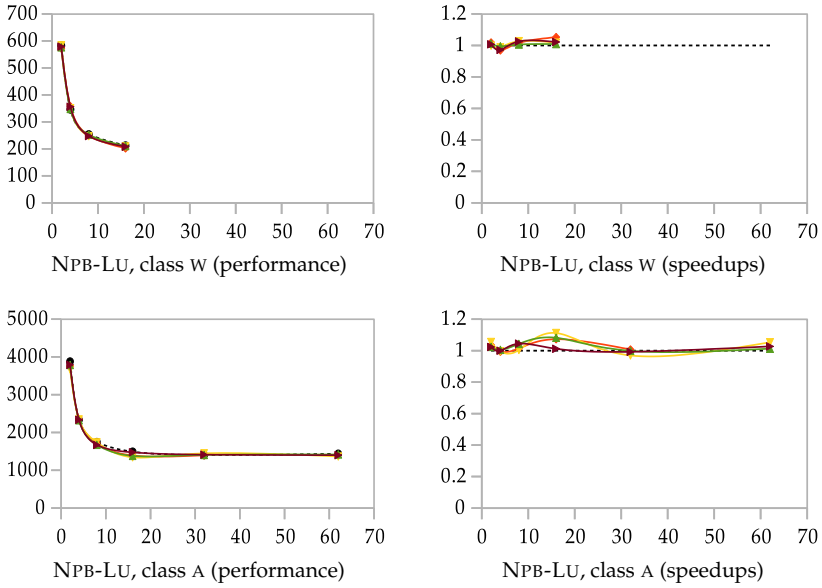


Figure 9.11: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

marks by Frumkin et al. [FSJY02, FSJY03]. In this realistic and extensive case study consisting of full Java programs instead of protocols in isolation, overall, the FOCAML-to-Java-compiled versions perform better with every successive improvement. Moreover, with all improvements in place, the FOCAML-to-Java-compiled versions perform roughly as well as the Java versions, sometimes a bit slower, but sometimes also a bit faster. If anything, these promising results indicate the practical feasibility of using a high, intention-expressing level of abstraction for implementing protocols. [Sections 4.2, 5.2, 6.2, and 7.2]

Although the current FOCAML compiler targets Java, neither its compiler-generated code nor the corresponding run-time library uses any Java-specific features. Combined with the fact that neither the Hybrid Approach, nor syntactic subtraction, nor commandification, nor queue-inference—all formalized and proven correct at the higher level of constraint automata instead of at the lower level of GPL code—depend in any way on Java, any GPL that supports some form of threading and mutual exclusion may serve as a target language for FOCAML compilation. For instance, for his MSc thesis [vdN15], Van de Nes developed a compiler that generates C code. [Section 4.2]

9.2 Future Work

This thesis covers only the bare essentials of FOCAML compilation, leaving plenty of room for interesting and challenging future work. I mentioned some of the minor opportunities (i.e., smaller projects, perhaps good for a single publication) throughout the text. In this final section, I want to discuss three major opportunities (i.e., larger projects, probably good for multiple publications), in no particular order.

- *Further compilation*

  Explicit user-threading APIs (as defined in Chapter 1) expose just enough details of the underlying hardware—but no more—for software engineers to get reasonable performance with a reasonable amount of effort. Perhaps if such APIs would expose more details, expert software engineers would squeeze even more performance out of the hardware; average software engineers, however, would require a disproportionate amount of effort to write programs even with just reasonable performance. Consequently, explicit user-threading APIs became the minimal abstraction that software engineers use to write programs, in general.

  In my specific context of FOCAML compilation, however, it seems less obvious that also compiler-generated code should invoke explicit user-threading APIs. After all, as I need to build a compiler only once anyway, it may very well pay off to put extra effort in generating code *below* explicit user-threading APIs, and perhaps even *below* the operating system's kernel threads, directly managing cores, to further improve performance.

  In one extreme incarnation of this approach, a FOCAML compiler generates assembly code. This, however, requires a lot of effort and assembly expertise from the designers of that compiler. Moreover, portability may suffer. A better option, therefore, seems the use of portable frameworks that give more control over cores than traditional explicit user-threading APIs do but not at the cost of having to generate assembly code.

  With Halle and Arbab, I did preliminary work in this direction [**J**HA14a, **J**HA14b], based on the *Proto Runtime Toolkit* (PRT) [Hal11, HC13], developed by Halle for his PhD thesis. PRT consists, among other components, of a run-time system for C code and APIs. On its start-up, the PRT run-time system seizes control of the available cores from the operating system, thereby gaining full responsibility for scheduling instructions onto those cores. These cores remain hidden from software engineers, though, through an API for managing PRT threads and a separate API for imposing custom scheduling policies. PRT-aware C code invokes the former API to instantiate units of parallelism, which the PRT run-time system subsequently schedules onto cores, without interference by the operating system. Bypassing the operating system in this way (including its rather heavy-weight scheduler), should result in better performance.

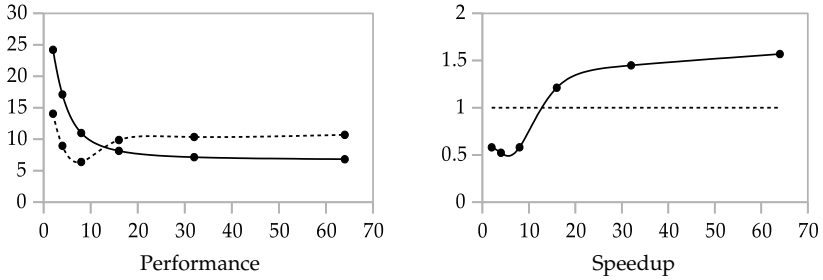


Figure 9.12: Left, performance (in thousands of cycles per `put`/`get` pair) as a function of the number of producers, denoted by k. Right, speedup relative to the dotted line as a function of k. Solid lines represent the performance/speedup of PRT-based compiler-generated code; dotted lines represent the performance/speedup of carefully optimized C+Pthreads-based hand-written code.

With Halle and Arbab, I performed preliminary experiments with a preliminary Reo-to-PRT compiler, and the preliminary results look encouraging [**J**HA14a, **J**HA14b]. For instance, Figure 9.12 shows the performance and speedup of two versions of a producers/consumer protocol for an increasing number of producers. For now, it remains unclear *exactly which* factors contribute to these promising results: perhaps PRT facilitates more efficient management of shared data structures (notably, queues) and/or context-switching between threads. Future research in this direction should target both the development of a more mature compiler and the conduction of more serious experiments to better understand and quantify the benefits of this approach.

- *Memory-centric compilation*

  In this thesis, I focused on the "algorithmic aspects" of FOCAML compilation, attempting to minimize the number of hardware instructions derived from compiler-generated code to reduce run time. As the experimental results for the NPB suite show, however, memory and caches seriously affect the run time of programs as well. Optimizing memory and cache usage constitutes a research field by itself, and for scientists working on those and related topics, this observation will not come as a surprise.

  Having built a foundation for the algorithmic aspects of FOCAML compilation, I firmly believe that future work should target improvements for optimizing memory and cache usage. Of course, at least code generated by the FOCAML-to-Java compiler should benefit from such improvements. However, I also foresee the need for a FOCAML-to-C compiler, to gain full control over memory allocation and make analyzing cache behavior easier, both of which the Java virtual machine obscures.

- *Type-aware compilation*

  In this thesis, I considered only untyped I/O operations: using the Java API for ports in Figure 1.9, every `put` sends an `Object`, while every `get` receives an `Object`. This untypedness makes worker code not only inelegant because of the many required **instanceof** checks and typecasts but also fragile: software engineers must know the type(s) of the objects returned by a `get` for every such invocation, and if they make a mistake, this goes unnoticed until their program crashes at run-time.

  Invented in the 1990s, *session types* seem a very suitable candidate for extending FOCAML with types [HVK98]. By annotating every port in a synchronization constraint in a constraint automaton with a type, this constraint automaton effectively becomes a global session type, amenable to projection on individual ports (as with multiparty session types [HYC08, CHY12]). The compiler can subsequently check the resulting per-port local session types against the actual usage of those ports in worker subprograms, to statically detect typing errors.

  Incidentally, with Santini and Arbab, I briefly sketched a different application of combining constraint automata with session types, namely projection of unprojectable choreographies [**J**SA15].

  Also incidentally, Ng et al. recently developed code generation technology for parametrized multiparty session types [NCY15], not unlike the work reported on in this thesis: using the work of Ng et al., software engineers implement protocols as global session types, after which a compiler generates MPI code and merges this code with existing computation code for workers. Though similar, the kinds of protocols supported by the DSL for interaction of Ng et al., called Pabble [NY14], seems limited compared to what FOCAML supports. For instance, FOCAML allows mixing synchronous and asynchronous interaction, which Pabble does not. Also, FOCAML has richer support for (functions and relations on) data. In contrast, Pabble better supports run-time parametrization, which FOCAML does not support whatsoever.

  In any case, the similarities between constraint automata and session types seem profound, and regardless of any practical motives, they require a better theoretical understanding. Future work in this direction should therefore target the development of constraint automata as global session types, both in theory and in practice.

The experimental results in this thesis show that compiler-generated code for intention-expressing protocol implementations can have performance comparable to hand-written code. I do not—and cannot—claim to achieve similar results in all possible scenarios, and much more experimentation and real-world case studies need to follow this initial piece of work. Nevertheless, by demonstrating promising first results, this thesis provides a justification for pursuing such future work. And that, perhaps, comprises the most valuable contribution of this thesis.

Abstract

English

In the early 2000s, hardware manufacturers shifted their attention from manufacturing faster—yet purely sequential—unicore processors to manufacturing slower—yet increasingly parallel—multicore processors. In the wake of this shift, parallel programming became essential for writing scalable programs on general hardware. Conceptually, every parallel program consists of workers, which implement primary units of sequential computation, and protocols, which implement the rules of interaction that workers must abide by. As programmers have been writing sequential code for decades, programming workers poses no new fundamental challenges. What is new—and notoriously difficult—is programming of protocols.

In this thesis, I study an approach to protocol programming where programmers implement their workers in an existing general-purpose language (GPL), while they implement their protocols in a complementary domain-specific language (DSL). DSLs for protocols enable programmers to express interaction among workers at a higher level of abstraction than the level of abstraction supported by today's GPLs, thereby addressing a number of protocol programming issues with today's GPLs. In particular, in this thesis, I develop a DSL for protocols based on a theory of formal automata and their languages. The specific automata that I consider, called constraint automata, have transition labels with a richer structure than alphabet symbols in classical automata theory. Exactly these richer transition labels make constraint automata suitable for modeling protocols.

Constraint automata constitute the (denotational) semantics of the DSL presented in this thesis. On top of this semantics, I use two complementary syntaxes: an existing graphical syntax (based on the coordination language Reo) and a novel textual syntax. The main contribution of this thesis, then, consists of a compiler and four of its optimizations, all formalized and proven correct at the semantic level of constraint automata, using bisimulation. In addition to these theoretical contributions, I also present an implementation of the compiler and its optimizations, which supports Java as the complementary GPL, as plugins for Eclipse. Nothing in the theory developed in this thesis depends on Java, though; any language that supports some form of threading

and mutual exclusion may serve as a target for compilation. To demonstrate the practical feasibility of the GPL+DSL approach to protocol programming, I study the performance of the implemented compiler and its optimizations through a number of experiments, including the Java version of the NAS Parallel Benchmarks. The experimental results in these benchmarks show that, with all four optimizations in place, compiler-generated protocol code can compete with hand-crafted protocol code.

A more extensive summary of this thesis appears in Chapter 9.

Nederlands

Sinds het begin van dit millennium hebben hardwarefabrikanten hun aandacht verschoven van het produceren van snellere—doch puur sequentiële—unicore processors ("éénkernige processoren") naar het produceren van langzamere—doch in toenemende mate parallelle—multicore processors ("meerkernige processoren"). Als een gevolg van deze verschuiving is parallel programmeren een essentieel onderdeel geworden van het schrijven van schaalbare programma's voor algemene hardware. Conceptueel gezien bestaat elk parallel programma uit arbeiders, die primair sequentiële berekeningen uitvoeren, en protocollen, die de interactieregels vastleggen waar arbeiders zich aan moeten conformeren. Aangezien programmeurs al decennialang sequentiële code schrijven zorgt het programmeren van arbeiders voor weinig nieuwe fundamentele uitdagingen. Wat wel nieuw is—en berucht in haar complexiteit—is het programmeren van protocollen.

In dit proefschrift onderzoek ik een methode voor het programmeren van protocollen waarin programmeurs arbeiders implementeren in een bestaande general-purpose language ("algemeen-gebruik taal"), afgekort GPL, terwijl zij protocollen implementeren in een domain-specific language ("domein-specifieke taal"), afgekort DSL. DSLs voor protocollen stellen programmeurs in staat interactie tussen arbeiders op een hoger abstractieniveau uit te drukken dan het abstractieniveau dat vandaag de dag wordt ondersteund door GPLs. Hierdoor word een aantal protocolprogrammeerproblemen van de huidige GPLs aangepakt. Om precies te zijn ontwikkel ik in dit proefschrift een DSL voor protocollen gebaseerd op een theorie van formele automaten en hun talen. De specieke automaten waar ik naar kijk, genaamd constraint automata ("beperkingsautomaten"), hebben transitielabels met een rijkere structuur dan alfabetsymbolen in de klassieke automatentheorie. Precies deze rijkere transitielabels maken constraint automata geschikt voor het modelleren van protocollen.

Constraint automata vormen de (denotationele) semantiek van de DSL die ik presenteer in dit proefschrift. Bovenop deze semantiek gebruik ik twee complementaire syntaxes: een bestaande grafische syntax (gebaseerd op de coordinatietaal Reo) en een nieuwe tekstuele syntax. De hoofdbijdrage van dit proefschrift bestaat uit een compiler en vier van haar optimalisaties, die ik allemaal formaliseer en correct bewijs op het semantische niveau van constraint automata door gebruik te maken van bisimulatie. Naast deze theoretische bij-

dragen presenteer ik ook een implementatie van de compiler en haar optimalisaties, die bestaat uit plugins voor Eclipse. Hoewel deze implementatie enkel Java ondersteunt als de complementaire GPL is niets aan de theorie in dit proefschrift Java-specifiek. Elke taal die threading en mutual exclusion ("wederzijdse uitsluiting") ondersteunt kan dienen als compilatiedoel. Om de praktische haalbaarheid van de GPL+DSL-aanpak te demonstreren bestudeer ik ten slotte de prestaties van de geïmplementeerde compiler en haar optimalisaties in een aantal experimenten, waaronder de Java-versie van de NAS Parallel Benchmarks. De resultaten van deze experimenten laten zien dat, wanneer alle vier optimalisaties in stelling worden gebracht, door de compiler gegenereerde protocolcode zich kan meten met handgeschreven protocolcode.

Hoofdstuk 9 geeft een uitgebreidere samenvatting van dit proefschrift.

Curriculum Vitae

Born in 1987 in Seoul, Republic of Korea

Education

- 1999–2005: Gymnasium, Gertrudiscollege Roosendaal
- 2005–2006: Propedeuse, Technische Universiteit Eindhoven
- 2006–2008: BSc (cum laude), Technische Universiteit Delft
- 2008–2010: MSc (cum laude), Technische Universiteit Delft

Positions

- 2010–2015: Doctoral candidate, Centrum Wiskunde & Informatica
- 2010–2015: Doctoral candidate, Universiteit Leiden
- 2015–...: Lecturer, Open Universiteit
- 2015–...: Guest researcher, Radboud Universiteit Nijmegen

Bibliography

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. The Landscape of Parallel Computing: A View from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, 2006.

(Pages 1 and 275.)

[ABD+09] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A View of the Parallel Computing Landscape. *Communications of the ACM*, 52(10):56–67, 2009.

doi:10.1145/1562764.1562783.

(Page 17.)

[ABdB+05] Farhad Arbab, Christel Baier, Frank de Boer, Jan Rutten, and Marjan Sirjani. Synthesis of Reo Circuits for Implementation of Component-Connector Automata Specifications. In *COORDINATION 2005*, volume 3454 of *LNCS*, pages 236–251. Springer, 2005.

doi:10.1007/11417019_16.

(Page 140.)

[ABdBR07] Farhad Arbab, Christel Baier, Frank de Boer, and Jan Rutten. Models and temporal logical specifications for timed component connectors. *Software & Systems Modeling*, 6(1):59–82, 2007.

doi:10.1007/s10270-006-0009-9.

(Page 31.)

[ACH05] Siva Anantharaman, Jing Chen, and Gaétan Hains. A Synchronous Process Calculus for Service Costs. In *SEFM 2005*, pages 435–444. IEEE, 2005.

doi:10.1109/SEFM.2005.6.

(Page 147.)

[AdBO09] Krzysztof Apt, Frank de Boer, and Ernst-Rüdiger Olderog. While Programs. In *Verification of Sequential and Concurrent Programs*, Texts in Computer Science, chapter 3, pages 55–126. Springer, 3rd edition, 2009.

doi:10.1007/978-1-84882-745-5_3.

(Pages 204, 205, 206, 207, and 280.)

[AFF01] Giorgio Ausiello, Paolo Franciosa, and Daniele Frigioni. Directed Hypergraphs: Problems, Algorithmic Results, and a Novel Decremental Approach. In *ICTCS 2001*, volume 2202 of *LNCS*, pages 312–328. Springer, 2001.

doi:10.1007/3-540-45446-2_20.

(Page 216.)

[AHS93] Farhad Arbab, Ivan Herman, and Per Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.

doi:10.1002/cpe.4330050103.

(Page 14.)

[AKM08] Farhad Arbab, Natallia Kokash, and Sun Meng. Towards Using Reo for Compliance-Aware Business Process Modeling. In *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008)*, volume 17 of *CCIS*, pages 108–123. Springer, 2008.

doi:10.1007/978-3-540-88479-8_9.

(Page 60.)

[AM08] Farhad Arbab and Sun Meng. Synthesis of Connectors from Scenario-Based Interaction Specifications. In *CBSE 2008*, volume 5282 of *LNCS*, pages 114–129. Springer, 2008.

doi:10.1007/978-3-540-87891-9_8.

(Page 60.)

[Amd67] Gene Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS SJCC 1967*, pages 483–485. ACM, 1967.

doi:10.1145/1465482.1465560.

(Pages 3 and 275.)

[Apt09a] Krzysztof Apt. Introduction. In *Principles of Constraint Programming*, chapter 1, pages 1–7. Cambridge University Press, 2nd edition, 2009.

(Pages 115 and 217.)

[Apt09b] Krzysztof Apt. Some Complete Constraint Solvers. In *Principles of Constraint Programming*, chapter 4, pages 82–134. Cambridge University Press, 2nd edition, 2009.

(Page 217.)

[AR03] Farhad Arbab and Jan Rutten. A Coinductive Calculus of Component Connectors. In *WADT 2002*, volume 2755 of *LNCS*, pages 34–55. Springer, 2003.

doi:10.1007/978-3-540-40020-2_2.

(Page 31.)

[Arb96] Farhad Arbab. The IWIM Model for Coordination of Concurrent Activities. In *COORDINATION 1996*, volume 1061 of *LNCS*, pages 34–56. Springer, 1996.

doi:10.1007/3-540-61052-9_38.

(Page 14.)

[Arb98] Farhad Arbab. What Do You Mean, Coordination? *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, pages 10–21, 1998.

(Page 14.)

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

doi:10.1017/S0960129504004153.

(Pages 18, 22, 51, 58, 60, and 277.)

[Arb05] Farhad Arbab. Abstract Behavior Types: a foundation model for components and their composition. *Science of Computer Programming*, 55(1–3):3–52, 2005.

doi:10.1016/j.scico.2004.05.010.

(Page 31.)

[Arb11] Farhad Arbab. Puff, The Magic Protocol. In *Talcott Festschrift*, volume 7000 of *LNCS*, pages 169–206. Springer, 2011.

doi:10.1007/978-3-642-24933-4_9.

(Pages 10, 13, 58, 60, and 277.)

[Atk10] Alistair Atkinson. A Dynamic, Decentralised Search Algorithm for Efficient Data Retrieval in a Distributed Tuple Space. In *AusPDC 2010*, pages 21–30. ACM, 2010.

(Page 141.)

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent Programming. In *Concurrent Programming in* ERLANG, chapter 5, pages 67–84. Prentice Hall, 2nd edition, 1996.

(Pages 5 and 275.)

[BB15] Eduard Baranov and Simon Bliudze. Offer semantics: Achieving compositionality, flattening and full expressiveness for the glue operators in BIP. *Science of Computer Programming*, 109:2–35, 2015.

doi:10.1016/j.scico.2015.05.011.

(Page 247.)

[BBB+91] David Bailey, Eric Barszcz, John Barton, David Browning, Russell Carter, Leonardo Dagum, Rod Fatoohi, Paul Frederickson, Thomas Lasinski, Robert Schreiber, Horst Simon, Venkat Venkatakrishnan, and Sisira Weeratunga. The Nas Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

doi:10.1177/109434209100500306.

(Pages 23, 80, and 285.)

[BBB+94] David Bailey, Eric Barszcz, John Barton, David Browning, Russell Carter, Leonardo Dagum, Rod Fatoohi, Paul Frederickson, Thomas Lasinski, Robert Schreiber, Horst Simon, Venkat Venkatakrishnan, and Sisira Weeratunga. The Nas Parallel Benchmarks. Technical Report RNR-94-007, NASA, 1994.

(Pages 23, 79, 80, 81, and 285.)

[BBJ+12] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, 25(5):383–409, 2012.

doi:10.1007/s00446-012-0168-6.

(Page 141.)

[BBK+10] Christel Baier, Tobias Blechmann, Joachim Klein, Sascha Klüppelholz, and Wolfgang Leister. Design and Verification of Systems with Exogenous Coordination Using Vereofy. In *ISoLA 2010*, volume 6416 of *LNCS*, pages 97–111. Springer, 2010.

doi:10.1007/978-3-642-16561-0_15.

(Pages 31, 38, and 69.)

[BBKK09a] Christel Baier, Tobias Blechmann, Joachim Klein, and Sascha Klüppelholz. A Uniform Framework for Modeling and Verifying Components and Connectors. In *COORDINATION 2009*, volume 5521 of *LNCS*, pages 247–267. Springer, 2009.

doi:10.1007/978-3-642-02053-7_13.

(Pages 31, 38, and 69.)

[BBKK09b] Christel Baier, Tobias Blechmann, Joachim Klein, and Sascha Klüppelholz. Formal Verification for Components and Connectors. In *FMCO 2008*, volume 5751 of *LNCS*, pages 82–101. Springer, 2009.

doi:10.1007/978-3-642-04167-9_5.

(Pages 31 and 69.)

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *SEFM 2006*, pages 3–12. IEEE, 2006.

doi:10.1109/SEFM.2006.27.

(Pages 14 and 141.)

[BCS09] Marcello Bonsangue, Dave Clarke, and Alexandra Silva. Automata for Context-Dependent Connectors. In *COORDINATION 2009*, volume 5521 of *LNCS*, pages 184–203. Springer, 2009.

doi:10.1007/978-3-642-02053-7_10.

(Page 246.)

[BCS12] Marcello Bonsangue, Dave Clarke, and Alexandra Silva. A model of context-dependent component connectors. *Science of Computer Programming*, 77(66):685–706, 2012.

doi:10.1016/j.scico.2011.01.006.

(Page 246.)

[Ben06] Mordechai Ben-Ari. Semaphores. In *Principles of Concurrent and Distributed Programming*, chapter 6, pages 107–144. Addison-Wesley, 2nd edition, 2006.

(Page 15.)

[Bjo93] Robert Bjornson. *Linda on Distributed Memory Multiprocessors*. PhD thesis, Yale University, 1993.

(Page 141.)

[BKK10] Tobias Blechmann, Joachim Klein, and Sascha Klüppelholz. *Vereofy V1.1 - User Manual*. Technische Universität Dresden, 2010.

(Pages 72 and 73.)

[BKK11] Christel Baier, Joachim Klein, and Sascha Klüppelholz. Modeling and Verification of Components and Connectors. In *SFM 2011*, volume 6659 of *LNCS*, pages 114–147. Springer, 2011.

doi:10.1007/978-3-642-21455-4_4.

(Pages 31 and 69.)

[BKK14] Christel Baier, Joachim Klein, and Sascha Klüppelholz. Synthesis of Reo Connectors for Strategies and Controllers. *Fundamenta Informatica*, 130(1):1–20, 2014.

doi:10.3233/FI-2014-980.

(Page 140.)

[BMFL02] Christian Bessière, Pedro Meseguer, Eugene Freuder, and Javier Larrosa. On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141(1–2):205–224, 2002.

doi:10.1016/S0004-3702(02)00263-1.

(Pages 115 and 217.)

[BO05] Miguel Bugalho and Arlindo Oliveira. Inference of regular languages using state merging algorithms with search. *Pattern Recognition*, 38(9):1457–1467, 2005.

doi:10.1016/j.patcog.2004.03.027.

(Page 247.)

[BS08] Simon Bliudze and Joseph Sifakis. The Algebra of Connectors—Structuring Interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315–1330, 2008.

doi:10.1109/TC.2008.26.

(Pages 38, 61, and 247.)

[BS10] Simon Bliudze and Joseph Sifakis. Causal semantics for the algebra of connectors. *Formal Methods in System Design*, 36(2):167–194, 2010.

doi:10.1007/s10703-010-0091-z.

(Pages 38, 61, and 247.)

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.

doi:10.1016/j.scico.2005.10.008.

(Pages 22, 38, 41, 48, 60, and 277.)

[BST89] Henri Bal, Jennifer Steiner, and Andrew Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

doi:10.1145/72551.72552.

(Page 95.)

[BV08] Jasper Berendsen and Frits Vaandrager. Compositional Abstraction in Real-Time Model Checking. In *FORMATS 2008*, volume 5215 of *LNCS*, pages 233–249. Springer, 2008.

doi:10.1007/978-3-540-85778-5_17.

(Page 147.)

[BvG87] Jos Baeten and Rob van Glabbeek. Merge and Termination in Process Algebra. In *FST&TCS 1987*, volume 287 of *LNCS*, pages 153–172. Springer, 1987.

doi:10.1007/3-540-18625-5_49.

(Page 147.)

[CBM+08] Călin Caşcaval, Colin Blundell, Maged Michael, Harold Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why is it Only a Research Toy? *Communications of the ACM*, 51(11):40–46, 2008.

doi:10.1145/1400214.1400228.

(Page 16.)

[CCA07] Dave Clarke, David Costa, and Farhad Arbab. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming*, 66(3):205–225, 2007.

doi:10.1016/j.scico.2007.01.009.

(Pages 99 and 137.)

[CHY12] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centered Programming for Web Services. *ACM Transactions on Programming Languages and Systems*, 34(2):8·1–8·78, 2012.

doi:10.1145/2220365.2220367.

(Page 294.)

[CKA10] Behnaz Changizi, Natallia Kokash, and Farhad Arbab. A Unified Toolset for Business Process Model Formalization. In *Preproceedings of FESCA 2010*, pages 147–156, 2010.

(Page 60.)

[CL00] Sung-Eun Choi and Christopher Lewis. A Study of Common Pitfalls in Simple Multi-Threaded Programs. *ACM SIGCSE Bulletin (SIGCSE 2000)*, 32(3):325–329, 2000.

doi:10.1145/330908.331879.

(Page 10.)

[CO94] Rafael Carrasco and Jose Oncina. Learning stochastic regular grammars by means of a state merging method. In *ICGI 1994*, volume 862 of *LNCS*, pages 139–152. Springer, 1994.

doi:10.1007/3-540-58473-0_144.

(Page 247.)

[Col88] Murray Cole. *Algorithmic Skeletons: a Structured Approach to the Management of Parallel Computation*. PhD thesis, University of Edinburgh, 1988.

(Pages 16 and 276.)

[CP12] Dave Clarke and José Proença. Partial Connector Colouring. In *COORDINATION 2012*, volume 7274 of *LNCS*, pages 59–73. Springer, 2012.

doi:10.1007/978-3-642-30829-1_5.

(Pages 99, 137, and 279.)

[CPLA11] Dave Clarke, José Proença, Alexander Lazovik, and Farhad Arbab. Channel-based coordination via constraint satisfaction. *Science of Computer Programming*, 76(8):681–710, 2011.

doi:10.1016/j.scico.2010.05.004.

(Pages 38, 99, and 220.)

[Dij82] Edsger Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, Texts and Monographs in Computer Science, chapter EWD447, pages 60–66. Springer, 1982.

doi:10.1007/978-1-4612-5695-3_12.

(Page 14.)

[Dij02] Edsger Dijkstra. Cooperating Sequential Processes. In *The Origin of Concurrent Programming*, chapter 1, pages 65–138. Springer, 2002.

doi:10.1007/978-1-4757-3472-0_2.

(Pages 6 and 275.)

[DJAB15] Kasper Dokter, **Sung-Shik Jongmans**, Farhad Arbab, and Simon Bliudze. Relating BIP and Reo. In *ICE 2015*, volume 189 of *EPTCS*, pages 3–20. CoRR, 2015.

doi:10.4204/EPTCS.189.3.

(Page 61.)

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

doi:10.1109/99.660313.

(Pages 5 and 275.)

[FGY94] Ming-Dong Feng, Yao-Qing Gao, and Chung-Kwong Yuen:. Distributed Linda Tuplespace Algorithms and Implementations. In *Parallel Processing: CONPAR 94 – VAPP VI*, volume 854 of *LNCS*, pages 581–592. Springer, 1994.

doi:10.1007/3-540-58430-7_51.

(Pages 140 and 141.)

[FSJY02] Michael Frumkin, Matthew Schultz, Haoqiang Jin, and Jerry Yan. Implementation of the NAS Parallel Benchmarks in Java. Technical Report NAS-02-009, NASA, 2002.

(Pages 80, 82, 83, and 291.)

[FSJY03] Michael Frumkin, Matthew Schultz, Haoqiang Jin, and Jerry Yan. Performance and Scalability of the NAS Parallel Benchmarks in Java. In *IPDPS 2003*, pages 139–44. IEEE, 2003.

doi:10.1109/IPDPS.2003.1213267.

(Pages 80, 285, and 291.)

[FWY96] Ming-Dong Feng, Weng-Fai Wong, and Chung-Kwong Yuen. BaLinda Lisp: Design and implementation. *Computer Languages*, 22(4):205–214, 1996.

doi:10.1016/S0096-0551(96)00016-1.

(Page 140.)

[Gel85] David Gelernter. Generative Communication in Linda. *Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

doi:10.1145/2363.2433.

(Page 140.)

[GL02] Dimitra Giannakopoulou and Flavio Lerda. From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In *FORTE 2002*, volume 2529 of *LNCS*, pages 308–326. Springer, 2002.

doi:10.1007/3-540-36135-9_20.

(Page 247.)

[GL10] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

doi:10.1002/spe.1026.

(Pages 17 and 276.)

[GLPN93] Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2–3):177–201, 1993.

doi:10.1016/0166-218X(93)90045-P.

(Page 213.)

[GPB+06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Task Execution. In *Java Concurrency in Practice*, chapter 6, pages 113–134. Addison-Wesley, 2006.

(Pages 5 and 275.)

[Hal11] Sean Halle. *A Study of Frameworks for Collectively Meeting the Productivity, Portability, and Adoptability Goals for Parallel Software*. PhD thesis, University of California, Santa Cruz, 2011.

(Page 292.)

[Hal12] Philipp Haller. On the Integration of the Actor Model in Mainstream Technologies. In *AGERE! 2012*, pages 1–6. ACM, 2012.

doi:10.1145/2414639.2414641.

(Pages 5 and 275.)

[HC13] Sean Halle and Albert Cohen. A Mutable Hardware Abstraction to Replace Threads. In *LCPC 2011*, volume 7146 of *LNCS*, pages 185–202. Springer, 2013.

doi:10.1007/978-3-642-36036-7_13.

(Page 292.)

[Her14] Maurice Herlihy. The Multicore Transformation. *Ubiquity*, 2014(9):1–9, 2014.

doi:10.1145/2618405.

(Page 16.)

[HM93] Maurice Herlihy and Eliot Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *ACM SIGARCH Computer Architecture News (ISCA 1993)*, 21(2):289–300, 1993.

doi:10.1145/173682.165164.

(Pages 16 and 276.)

[HMU06] John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. Pushdown Automata. In *Introduction to Automata Theory, Languages and Computation*, chapter 6, pages 225–260. Addison-Wesley, 3rd edition, 2006.

(Page 39.)

[HO09] Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.

doi:10.1016/j.tcs.2008.09.019.

(Pages 5 and 275.)

[Hoa69] Tony Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

doi:10.1145/363235.363259.

(Pages 205 and 280.)

[Hoa74] Tony Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

doi:10.1145/355620.361161.

(Pages 6 and 275.)

[HP11a] John Hennessy and David Patterson. Instruction-Level Parallelism and Its Exploitation. In *Computer Architecture: A Quantitative Approach*, chapter 3, pages 145–259. Elsevier, 5th edition, 2011.

(Page 1.)

[HP11b] John Hennessy and David Patterson. Memory Hierarchy Design. In *Computer Architecture: A Quantitative Approach*, chapter 2, pages 69–144. Elsevier, 5th edition, 2011.

(Page 165.)

[HP11c] John Hennessy and David Patterson. Vector Processors. In *Computer Architecture: A Quantitative Approach*, chapter G, pages G·1–G·34. Elsevier, 5th edition, 2011.

(Page 80.)

[HVK98] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP 1998*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

doi:10.1007/BFb0053567.

(Page 294.)

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *ACM SIGPLAN Notices (POPL 2008)*, 43(1):273–284, 2008.

doi:10.1145/1328897.1328472.

(Page 294.)

[IBC11] Mohammad Izadi, Marcello Bonsangue, and Dave Clarke. Büchi automata for modeling component connectors. *Software & Systems Modeling*, 10(2):183–200, 2011.

doi:10.1007/s10270-010-0152-1.

(Page 31.)

[Iza11] Mohammad Izadi. *Model Checking of Component Connectors*. PhD thesis, Universiteit Leiden, 2011.

hdl:1887/18189.

(Page 31.)

[**J**A11] **Sung-Shik Jongmans** and Farhad Arbab. Correlating Semantic Models of Reo Connectors: Connector Coloring and Constraint Automata. In *ICE 2011*, volume 59 of *EPTCS*, pages 84–103. CoRR, 2011.

doi:10.4204/EPTCS.59.8.

(Page 25.)

[**J**A12] **Sung-Shik Jongmans** and Farhad Arbab. Overview of Thirty Semantic Formalisms for Reo. *Scientific Annals of Computer Science*, 22(1):201–251, 2012.

doi:10.7561/SACS.2012.1.201.

(Pages 25 and 60.)

[**J**A13a] **Sung-Shik Jongmans** and Farhad Arbab. Global Consensus through Local Synchronization. In *Advances in Service-Oriented and Cloud Computing (FOCLASA 2013)*, volume 393 of *CCIS*, pages 174–188. Springer, 2013.

doi:10.1007/978-3-642-45364-9_15.

(Pages 25 and 127.)

[**J**A13b] **Sung-Shik Jongmans** and Farhad Arbab. Modularizing and Specifying Protocols among Threads. In *PLACES 2012*, volume 109 of *EPTCS*, pages 34–45. CoRR, 2013.

doi:10.4204/EPTCS.109.6.

(Pages 25 and 95.)

[JA14] **Sung-Shik Jongmans** and Farhad Arbab. Toward Sequentializing Overparallelized Protocol Code. In *ICE 2014*, volume 166 of *EPTCS*, pages 38–44. CoRR, 2014.

(Pages 25, 95, and 127.)

[JA15a] **Sung-Shik Jongmans** and Farhad Arbab. Can High Throughput Atone for High Latency in Compiler-Generated Protocol Code? In *FSEN 2015*, volume 9392 of *LNCS*, pages 238–258. Springer, 2015.

doi:10.1007/978-3-319-24644-4_17.

(Page 25.)

[JA15b] **Sung-Shik Jongmans** and Farhad Arbab. Take Command of Your Constraints! In *COORDINATION 2015*, volume 9037 of *LNCS*, pages 117–132. Springer, 2015.

doi:10.1007/978-3-319-19282-6_8.

(Pages 25 and 203.)

[JA16] **Sung-Shik Jongmans** and Farhad Arbab. Global consensus through local synchronization: A formal basis for partially-distributed coordination. *Science of Computer Programming*, 115–116:199–224, 2016.

doi:10.1016/j.scico.2015.09.001.

(Pages 25, 95, and 127.)

[JCP12] **Sung-Shik Jongmans**, Dave Clarke, and Jose Proença. A Procedure for Splitting Processes and its Application to Coordination. In *FOCLASA 2012*, volume 91 of *EPTCS*, pages 79–96. CoRR, 2012.

doi:10.4204/EPTCS.91.6.

(Pages 25, 138, and 279.)

[JCP16] **Sung-Shik Jongmans**, Dave Clarke, and Jose Proença. A procedure for splitting data-aware processes and its application to coordination. *Science of Computer Programming*, 115–116:47–78, 2016.

doi:10.1016/j.scico.2014.02.017.

(Pages 25, 138, and 279.)

[JFY99] Haoqiang Jin, Michael Frumkin, and Jerry Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report NAS-99-011, NASA, 1999.

(Page 80.)

[JHA14a] **Sung-Shik Jongmans**, Sean Halle, and Farhad Arbab. Automata-based Optimization of Interaction Protocols for Scalable Multicore Platforms. In *COORDINATION 2014*, volume 8459 of *LNCS*, pages 65–82. Springer, 2014.

doi:10.1007/978-3-662-43376-8_5.

(Pages 25, 239, 246, 292, and 293.)

[**J**HA14b] **Sung-Shik Jongmans**, Sean Halle, and Farhad Arbab. Reo: A Dataflow Inspired Language for Multicore. In *DFM 2013*, pages 42–50. IEEE, 2014.

doi:10.1109/DFM.2013.14 .

(Pages 25, 292, and 293.)

[**J**KA11] **Sung-Shik Jongmans**, Christian Krause, and Farhad Arbab. Encoding Context-Sensitivity in Reo into Non-Context-Sensitive Semantic Models. In *COORDINATION 2011*, volume 6721 of *LNCS*, pages 31–48. Springer, 2011.

doi:10.1007/978-3-642-21464-6_3.

(Pages 25 and 247.)

[**J**KA16] **Sung-Shik Jongmans**, Tobias Kappé, and Farhad Arbab. Composing Constraint Automata, State-by-State. In *FACS 2015*, volume 9539 of *LNCS*, chapter 12. Springer, 2016.

doi:10.1007/978-3-319-28934-2_12.

(Page 25.)

[**Jon**16] **Sung-Shik Jongmans**. Automata-Theoretic Protocol Programming (With Proofs). Technical Report FM-1601, Centrum Wiskunde & Informatica, 2016.

urn:nbn:nl:ui:18-24063.

(Pages 25, 61, 62, and 67.)

[**J**SA14] **Sung-Shik Jongmans**, Francesco Santini, and Farhad Arbab. Partially-Distributed Coordination with Reo. In *PDP 2014*, pages 697–706. IEEE, 2014.

doi:10.1109/PDP.2014.19.

(Pages 25 and 127.)

[**J**SA15] **Sung-Shik Jongmans**, Francesco Santini, and Farhad Arbab. Partially-Distributed Coordination with Reo and Constraint Automata. *Service Oriented Computing and Applications*, 9(3):311–339, 2015.

doi:10.1007/s11761-015-0177-y.

(Pages 25, 127, and 294.)

[**J**SS+12] **Sung-Shik Jongmans**, Francesco Santini, Mahdi Sargolzaei, Farhad Arbab, and Hamideh Afsarmanesh. Automatic Code Generation for the Orchestration of Web Services with Reo. In *ESOCC 2012*, volume 7592 of *LNCS*, pages 1–16. Springer, 2012.

doi:10.1007/978-3-642-33427-6_1.

(Page 25.)

[JSS+14] **Sung-Shik Jongmans**, Francesco Santini, Mahdi Sargolzaei, Farhad Arbab, and Hamideh Afsarmanesh. Orchestrating Web Services using Reo: From Circuits and Behaviors to Automatically Generated Code. *Service Oriented Computing and Applications*, 8(4):277–297, 2014.

doi:10.1007/s11761-013-0147-1.

(Page 25.)

[Kah62] Arthur Kahn. Topological Sorting in Large Networks. *Communications of the ACM*, 5(11):558–562, 1962.

doi:10.1145/368996.369025.

(Page 209.)

[KB09] Sascha Klüppelholz and Christel Baier. Symbolic model checking for channel-based component connectors. *Science of Computer Programming*, 74(9):688–701, 2009.

doi:10.1016/j.scico.2008.09.020.

(Pages 31, 38, and 69.)

[KB10] Sascha Klüppelholz and Christel Baier. Alternating-time stream logic for multi-agent systems. *Science of Computer Programming*, 75(6):398–425, 2010.

doi:10.1016/j.scico.2009.07.007.

(Page 31.)

[KC09] Christian Koehler and Dave Clarke. Decomposing Port Automata. In *SAC 2009*, pages 1369–1373. ACM, 2009.

doi:10.1145/1529282.1529587.

(Pages 38 and 140.)

[KGdV13] Christian Krause, Holger Giese, and Erik de Vink. Compositional and behavior-preserving reconfiguration of component connectors in Reo. *Journal of Visual Languages & Computing*, 24(3):153–168, 2013.

doi:10.1016/j.jvlc.2012.09.002.

(Pages 38 and 48.)

[KKSB11] Joachim Klein, Sascha Klüppelholz, Andries Stam, and Christel Baier. Hierarchical Modeling and Formal Verification. An Industrial Case Study Using Reo and Vereofy. In *FMICS 2011*, volume 6959 of *LNCS*, pages 228–243. Springer, 2011.

doi:10.1007/978-3-642-24431-5_17.

(Page 69.)

[Kla10] Martin Klazar. Some general results in combinatorial enumeration. In *Permutation Patterns*, volume 376 of *London Mathematical Society Lecture Note Series*, pages 3–40. Cambridge University Press, 2010.

(Page 130.)

[Kle12] Joachim Klein. *Compositional Synthesis and Most General Controllers*. PhD thesis, Technische Universität Dresden, 2012.

urn:nbn:de:bsz:14-qucosa-130654.

(Page 31.)

[Klü12] Sascha Klüppelholz. *Verification of Branching-Time and Alternating-Time Properties for Exogenous Coordination Models*. PhD thesis, Technische Universität Dresden, 2012.

urn:nbn:de:bsz:14-qucosa-86211.

(Pages 31, 38, 48, and 69.)

[KM03] Dietrich Kuske and Ingmar Meinecke. Branching Automata with Costs — A Way of Reflecting Parallelism in Costs. In *CIAA 2003*, volume 2759 of *LNCS*, pages 150–162. Springer, 2003.

doi:10.1007/3-540-45089-0_15.

(Page 149.)

[Kni86] Tom Knight. An Architecture for Mostly Functional Languages. In *LFP 1986*, pages 105–112. ACM, 1986.

doi:10.1145/319838.319854.

(Pages 16 and 276.)

[Knu97] Donald Knuth. Information Structures. In *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, chapter 2, pages 232–465. Addison-Wesley, 3rd edition, 1997.

(Page 209.)

[KQCM09] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. The Orc Programming Language. In *FMOODS/FORTE 2009*, volume 5522 of *LNCS*, pages 1–25. Springer, 2009.

doi:10.1007/978-3-642-02138-1_1.

(Page 149.)

[KS08] Bartek Klin and Vladimiro Sassone. Structural Operational Semantics for Stochastic Process Calculi. In *FOSSACS 2008*, volume 4962 of *LNCS*, pages 428–442. Springer, 2008.

doi:10.1007/978-3-540-78499-9_30.

(Page 148.)

[KV08] Bernhard Korte and Jens Vygen. Spanning Trees and Arborescences. In *Combinatorial Optimization: Theory and Algorithms*, volume 21 of *Algorithms and Combinatorics*, chapter 6, pages 127–150. Springer, 4th edition, 2008.

doi:10.1007/978-3-540-71844-4.

(Page 213.)

[Lam86] Leslie Lamport. The Mutual Exclusion Problem: Part I—A Theory of Interprocess Communication. *Journal of the ACM*, 33(2):313–326, 1986.

doi:10.1145/5383.5384.

(Page 6.)

[Lea11] Diana Lea. The Dictionary. In *Oxford Advanced American Dictionary*, pages 1–1736. Oxford University Press, 2011.

(Page 27.)

[Lee06] Edward Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.

doi:10.1109/MC.2006.180.

(Pages 6 and 10.)

[LNZ14] Edward Lee, Stephen Neuendorffer, and Gang Zhou. Synchronous-Reactive Models. In *System Design, Modeling, and Simulation using Ptolemy II*, chapter 5, pages 158–185. Ptolemy.org, 1st edition, 2014.

(Page 220.)

[LPP98] Kevin Lang, Barak Pearlmutter, and Rodney Price. Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm. In *Grammatical Inference*, volume 1433 of *LNCS*, pages 1–12. Springer, 1998.

doi:10.1007/BFb0054059.

(Page 247.)

[LSB09] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The Design of a Task Parallel Library. *ACM SIGPLAN Notices (OOPSLA 2009)*, 44(10):227–242, 2009.

doi:10.1145/1639949.1640106.

(Pages 5 and 275.)

[MAB11] Sun Meng, Farhad Arbab, and Christel Baier. Synthesis of Reo circuits from scenario-based interaction specifications. *Science of Computer Programming*, 76(8):651–680, 2011.

doi:10.1016/j.scico.2010.03.002.

(Page 60.)

[Mil89] Robin Milner. Strong Bisimulation and Strong Equivalence. In *Communication and Concurrency*, chapter 4, pages 84–105. Prentice Hall, 1989.

(Pages 40 and 277.)

[Moo98] Gordon Moore. Cramming More Components onto Integrated Circuits. *the IEEE*, 86(1):114–117, 1998.

doi:10.1109/JPROC.1998.658762.

(Page 1.)

[MRR12] Michael McCool, Arch Robinson, and James Reinders. Introduction. In *Structured Parallel Programming*, chapter 1, pages 1–38. Elsevier, 2012.

(Page 17.)

[MSM05] Timothy Mattson, Beverly Sanders, and Berna Massingill. A Pattern Language for Parallel Programming. In *Patterns for Parallel Programming*, SPS, chapter 1, pages 1–6. Addison-Wesley, 2005.

(Page 17.)

[MT03] Ronaldo Menezes and Robert Tolksdorf. A New Approach to Scalable Linda-systems Based on Swarms. In *SAC 2003*, pages 375–379. ACM, 2003.

doi:10.1145/952532.952607.

(Page 141.)

[NCY15] Nicholas Ng, Jose Coutinho, and Nobuko Yoshida. Protocols by Default: Safe MPI Code Generation based on Session Types. In *CC 2015*, volume 9031 of *LNCS*, pages 212–232. Springer, 2015.

doi:10.1007/978-3-662-46663-6_11.

(Page 294.)

[NY14] Nicholas Ng and Nobuko Yoshida. Pabble: Parameterised Scribble for Parallel Programming. In *PDP 2014*, pages 707–714. IEEE, 2014.

doi:10.1109/PDP.2014.20.

(Page 294.)

[PA98] George Papadopoulos and Farhad Arbab. Coordination Models and Languages. *Advances in Computers*, 46:329–400, 1998.

doi:10.1016/S0065-2458(08)60208-9.

(Page 141.)

[Par72] David Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

doi:10.1145/361598.361623.

(Page 14.)

[PC13a] José Proença and Dave Clarke. Data Abstraction in Coordination Constraints. In *Advances in Service-Oriented and Cloud Computing (FOCLASA 2013)*, volume 393 of *CCIS*, pages 159–173. Springer, 2013.

doi:10.1007/978-3-642-45364-9_14.

(Pages 38, 99, and 220.)

[PC13b] José Proença and Dave Clarke. Interactive Interaction Constraints. In *COORDINATION 2013*, volume 7890 of *LNCS*, pages 211–225. Springer, 2013.

doi:10.1007/978-3-642-38493-6_15.

(Pages 38, 99, and 220.)

[PCdVA11] José Proença, Dave Clarke, Erik de Vink, and Farhad Arbab. Decoupled execution of synchronous coordination models via behavioural automata. In *FOCLASA 2011*, volume 58 of *EPTCS*, pages 65–79. CoRR, 2011.

doi:10.4204/EPTCS.58.5.

(Pages 99, 137, and 279.)

[PCdVA12] José Proença, Dave Clarke, Erik de Vink, and Farhad Arbab. Dreams: a framework for distributed synchronous coordination. In *SAC 2012*, pages 1510–1515. ACM, 2012.

doi:10.1145/2245276.2232017.

(Pages 99, 137, and 279.)

[Pos14] Raphael Poss. Multicore architectures and their software landscape. In *Computer Handbook*, chapter 24, pages 24·1–24·17. CRC Press, 3rd edition, 2014.

(Pages 1 and 8.)

[Pro11] José Proença. *Synchronous Coordination of Distributed Components*. PhD thesis, Universiteit Leiden, 2011.

hdl:1887/17624.

(Pages 38, 99, 137, and 279.)

[PSAB12] Bahman Pourvatan, Marjan Sirjani, Farhad Arbab, and Marcello Bonsangue. Decomposition of Constraint Automata. In *FACS 2010*, volume 6921 of *LNCS*, pages 237–258. Springer, 2012.

doi:10.1007/978-3-642-27269-1_14.

(Pages 38, 48, and 140.)

[PSHA12] Bahman Pourvatan, Marjan Sirjani, Hossein Hojjat, and Farhad Arbab. Symbolic execution of Reo circuits using constraint automata. *Science of Computer Programming*, 77(7–8):848–869, 2012.

doi:10.1016/j.scico.2011.04.001.

(Pages 38 and 48.)

[Rau10a] Wolfgang Rautenberg. First-Order Logic. In *A Concise Introduction to Mathematical Logic*, Universitext, chapter 2, pages 41–90. Springer, 3rd edition, 2010.

doi:10.1007/978-1-4419-1221-3_2.

(Pages 32, 33, 36, 176, and 254.)

[Rau10b] Wolfgang Rautenberg. Propositional Logic. In *A Concise Introduction to Mathematical Logic*, Universitext, chapter 1, pages 1–40. Springer, 3rd edition, 2010.

doi:10.1007/978-1-4419-1221-3_1.

(Page 177.)

[Rei85] Wolfgang Reisig. Introductory Examples and Basic Definitions. In *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*, chapter 1, pages 3–16. Springer, 1985.

(Page 33.)

[Rei07] James Reinders. Why Threading Building Blocks? In *Intel Threading Building Blocks*, chapter 1, pages 1–6. O'Reilly, 2007.

(Pages 5 and 275.)

[Rob13] Arch Robison. Composable Parallel Patterns with Intel Cilk Plus. *Computing in Science and Engineering*, 15(2):66–71, 2013.

doi:10.1109/MCSE.2013.21.

(Pages 5 and 275.)

[RW96] Antony Rowstron and Alan Wood. An Efficient Distributed Tuple Space Implementation for Networks of Workstations. In *Euro-Par 1996*, volume 1123 of *LNCS*, pages 510–513. Springer, 1996.

doi:10.1007/3-540-61626-8_69.

(Page 141.)

[Seg95] Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Massachusetts Institute of Technology, 1995.

(Page 147.)

[SGG13] Abraham Silberschatz, Peter Galvin, and Greg Gagne. Threads. In *Operating System Concepts*, chapter 4, pages 163–201. Wiley, 9th edition, 2013.

(Page 8.)

[SJBA06] Marjan Sirjani, Mohammad-Mahdi Jaghoori, Christel Baier, and Farhad Arbab. Compositional Semantics of an Actor-Based Language Using Constraint Automata. In *COORDINATION 2006*, volume 4038 of *LNCS*, pages 281–297. Springer, 2006.

doi:10.1007/11767954_18.

(Page 65.)

[SO93] Andreas Stolcke and Stephen Omohundro. Hidden Markov Model Induction by Bayesian Model Merging. In *NIPS 1992*, pages 11–18. Morgan-Kaufmann, 1993.

(Page 247.)

[ST97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

doi:10.1007/s004460050028.

(Pages 16 and 276.)

[Sut05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs Journal*, 30(3), 2005.

(Page 1.)

[TDJ13] Samira Tasharofi, Peter Dinges, and Ralph Johnson. Why Do Scala Developers Mix the Actor Model with other Concurrency Models? In *ECOOP 2013*, volume 7920 of *LNCS*, pages 302–326. Springer, 2013.

doi:10.1007/978-3-642-39038-8_13.

(Pages 9 and 275.)

[TVMS08] Samira Tasharofi, Mohsen Vakilian, Roshanak Zilouchian Moghaddam, and Marjan Sirjani. Modeling Web Service Interactions Using the Coordination Language Reo. In *WS-FM 2007*, volume 4937 of *LNCS*, pages 108–123. Springer, 2008.

doi:10.1007/978-3-540-79230-7_8.

(Page 60.)

[Vaj11] András Vajda. Practical Many-Core Programming. In *Programming Many-Core Chips*, chapter 9, pages 175–211. Springer, 2011.

doi:10.1007/978-1-4419-9739-5_9.

(Page 8.)

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

doi:10.1145/352029.352035.

(Page 17.)

[vdN15] Mathijs van de Nes. In preparation. Master's thesis, Universiteit Leiden, 2015.

(Pages 20, 83, and 291.)

[Vra97] Jos Vrancken. The algebra of communicating processes with empty process. *Theoretical Computer Science*, 177(2):287–328, 1997.

doi:10.1016/S0304-3975(96)00250-2.

(Page 147.)

[WM95] William Wulf and Sally McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

doi:10.1145/216585.216588.

(Page 1.)

[WMLF98] Peter Wyckoff, Stephen McLaughry, Tobin Lehman, and Daniel Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

doi:10.1147/sj.373.0454.

(Page 140.)

[Woe92] Gerhard Woeginger. The complexity of finding arborescences in hypergraphs. *Information Processing Letters*, 44(3):161–164, 1992.

doi:10.1016/0020-0190(92)90057-3.

(Page 216.)

[Yeh93] Wei-Jen Yeh. *Controlling State Explosion in Reachability Analysis*. PhD thesis, Purdue University, 1993.

(Page 148.)

[YMG14] Leonid Yavits, Amir Morad, and Ran Ginosar. The effect of communication and synchronization on AmdahlâĂŹs law in multicore systems. *Parallel Computing*, 40(1):1–16, 2014.

doi:10.1016/j.parco.2013.11.001.

(Pages 3 and 275.)

Index

Titles in the IPA Dissertation Series since 2013

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking*. Faculty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems*. Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities*. Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2014-12

N. Noroozi. *Improving Input-Output Conformance Testing Theories*. Faculty of Mathematics and Computer Science, TU/e. 2014-13

M. Helvensteijn. *Abstract Delta Modeling: Software Product Lines and Beyond*. Faculty of Mathematics and Natural Sciences, UL. 2014-14

P. Vullers. *Efficient Implementations of Attribute-based Credentials on Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2014-15

F.W. Takes. *Algorithms for Analyzing and Mining Real-World Graphs*. Faculty of Mathematics and Natural Sciences, UL. 2014-16

M.P. Schraagen. *Aspects of Record Linkage*. Faculty of Mathematics and Natural Sciences, UL. 2014-17

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle*. Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking*. Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography*. Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems*. Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology*. Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction*. Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things*. Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes*. Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq*. Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01