Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/38223 holds various files of this Leiden University dissertation

Author: Jongmans, Sung-Shik T.Q.
Title: Automata-theoretic protocol programming : parallel computation, threads and their interaction, optimized compilation, [at a] high level of abstraction
Issue Date: 2016-03-03

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-

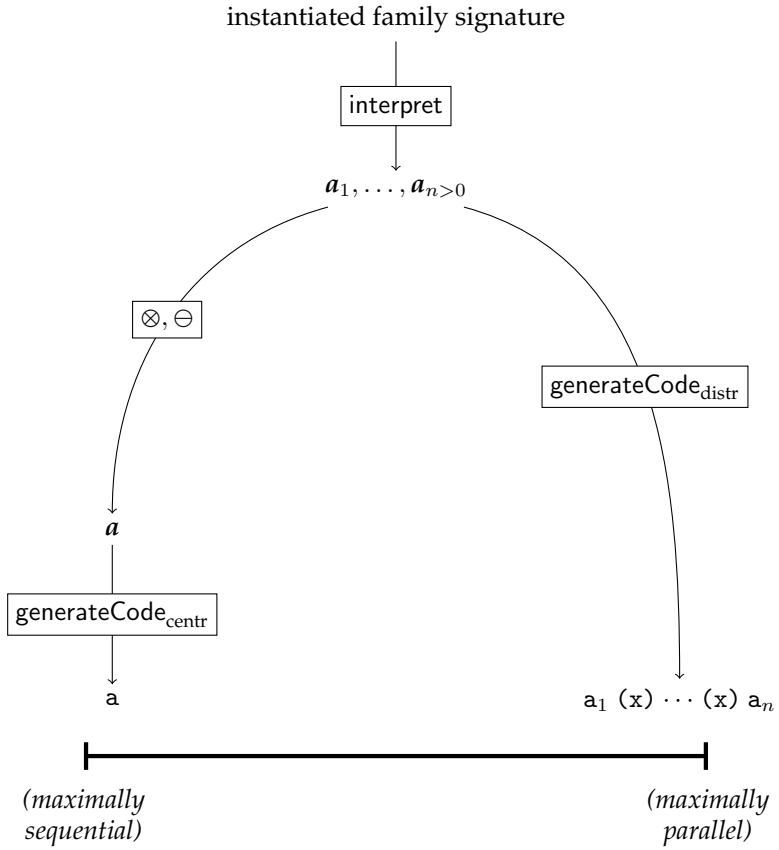

Figure 4.1: Basic compilation approaches

```
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

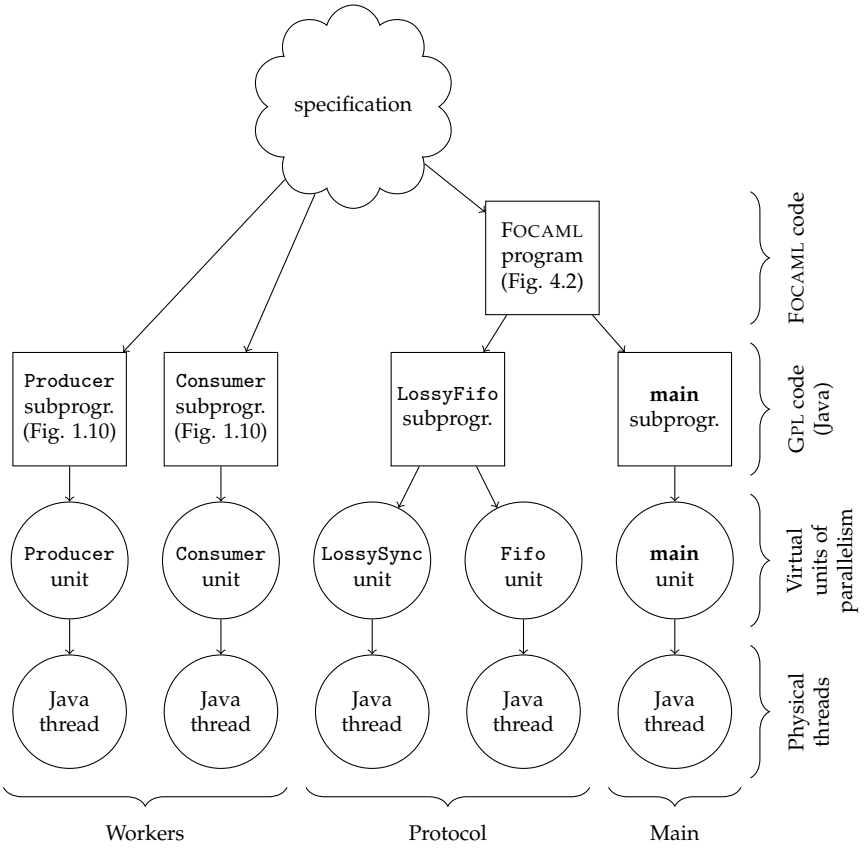

Figure 4.3: Relation between passive objects (squares) and active objects (circles) in the Distributed Approach on input of the FOCAML program in Figure 4.2

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-

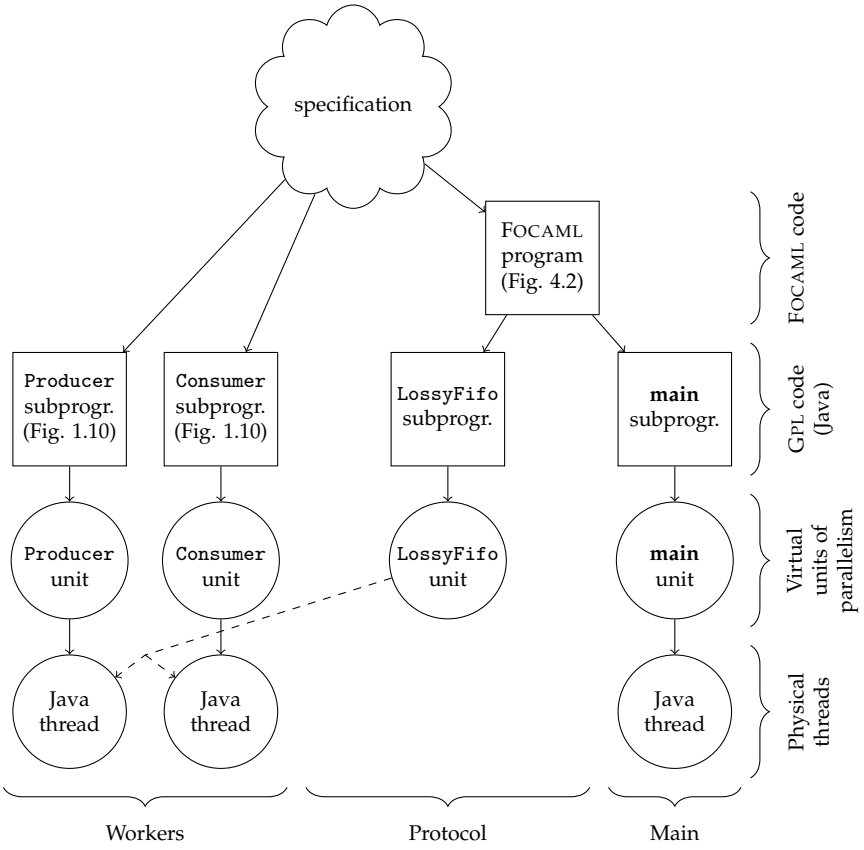

Figure 4.4: Relation between passive objects (squares) and active objects (circles) in the Centralized Approach on input of the FOCAML program in Figure 4.2. Dashed arrows indicate that the `LossyFifo` unit does not map to its own separate thread but "piggybacks" on the threads of the two worker units,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

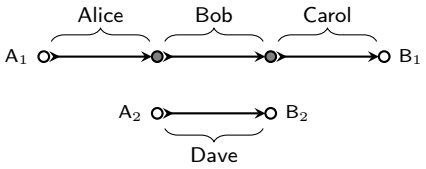

Figure 4.8: Alice, Bob, Carol, and Dave in Reo

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:

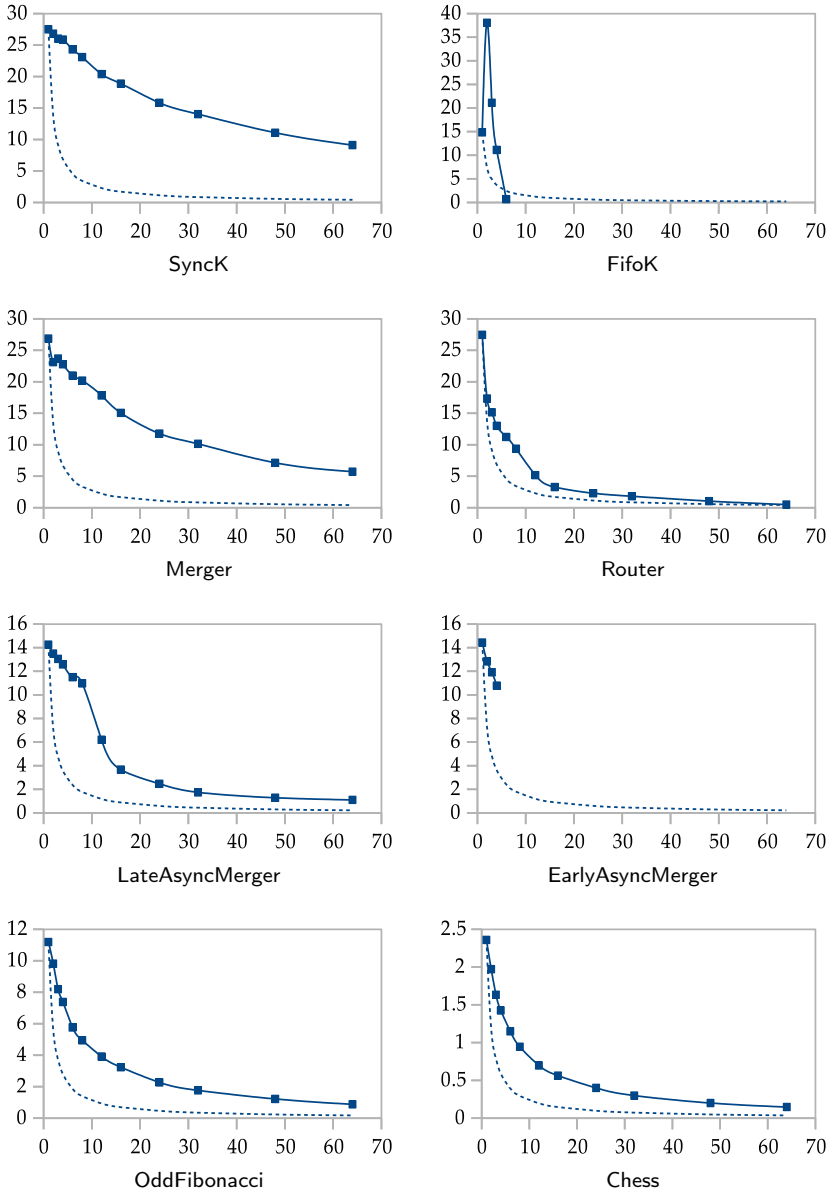

Figure 4.26: Performance (in number of completed rounds per four minutes) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See also the legend in Figure 9.1.

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.