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.

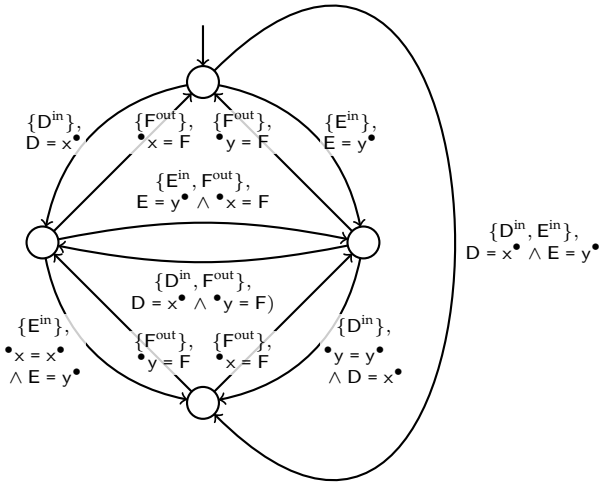


Figure 3.1: Constraint automaton for the EarlyAsyncMerger2 protocol. One producer has access to port D, the other producer has access to port E, the consumer has access to port F, and the producers and the consumer use buffers x and y for temporary storage of data.

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

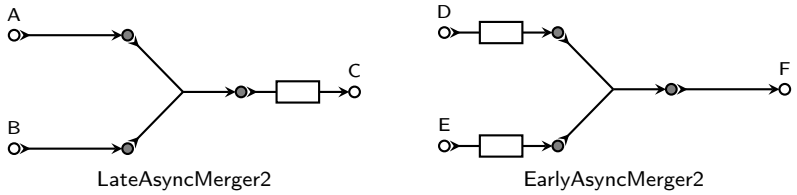


Figure 3.2: Graphical syntax for the constraint automata in Figures 2.2 and 3.1

```
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

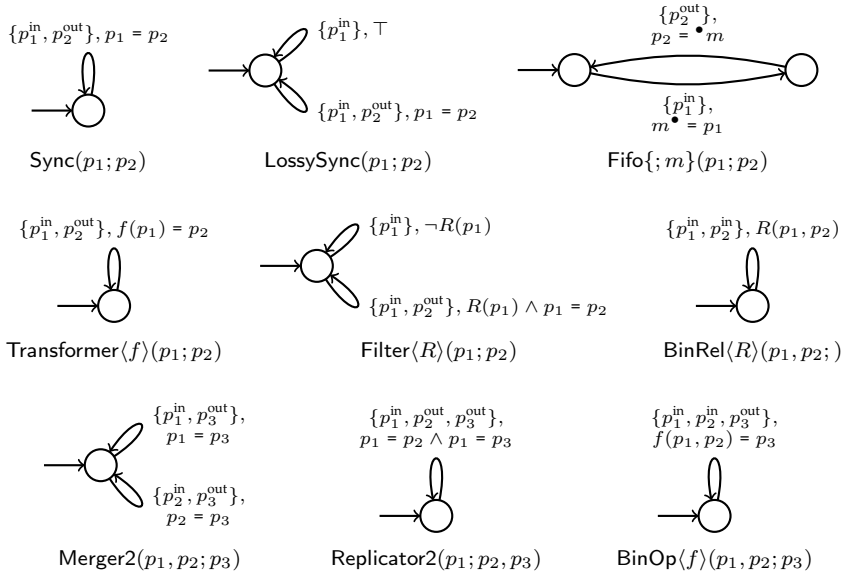


Figure 3.4: Parametric constraint automata in the core set

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

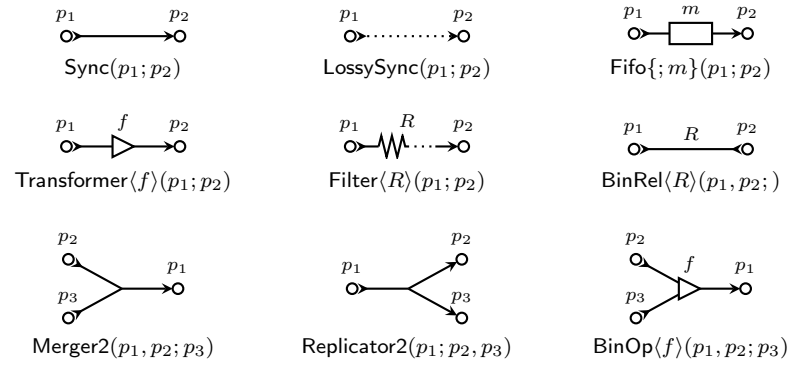


Figure 3.6: Hyperarcs for the families defined in Figure 3.4

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,

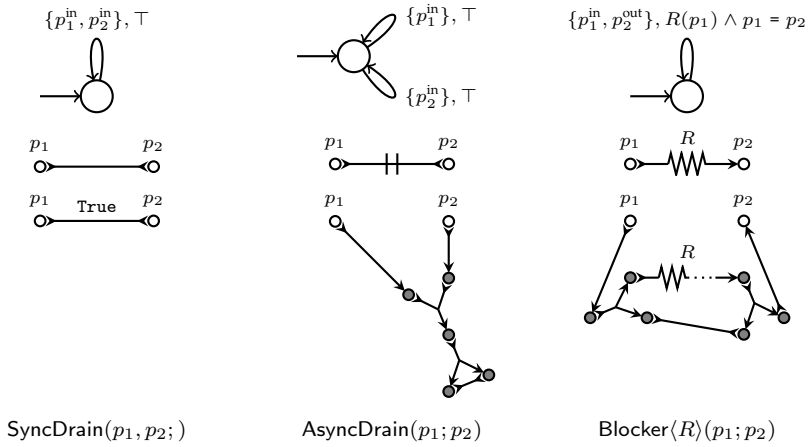


Figure 3.7: Hypergraphs, shorthands, and parametric constraint automata for families SyncDrain, AsyncDrain, and Blocker

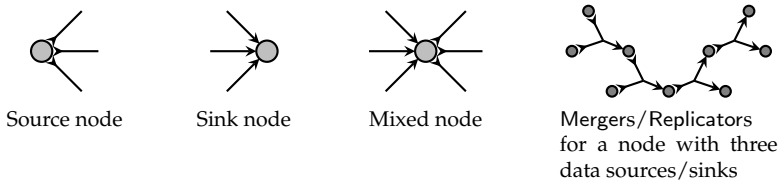


Figure 3.8: Nodes

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.

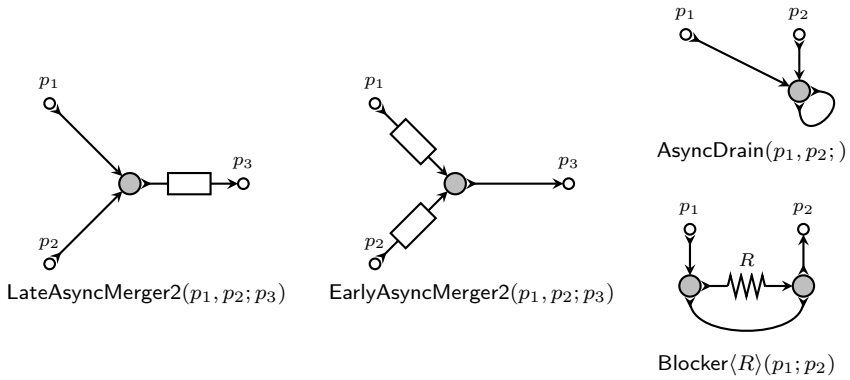


Figure 3.9: Circuits for families LateAsyncMerger2, EarlyAsyncMerger2, AsyncDrain, and Blocker (cf. Figures 3.2 and 3.7)

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

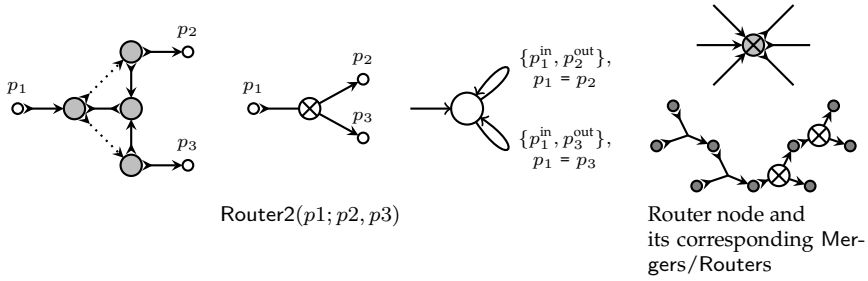


Figure 3.10: Circuit, shorthand, and parametric constraint automaton for family Router2 (left) and router node (right)

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-

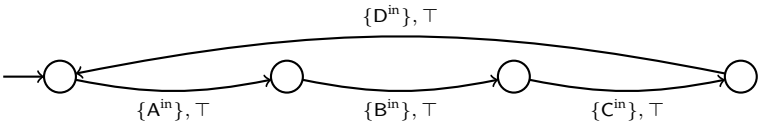


Figure 3.21: Constraint automaton for the $\mathsf{Sequencer}_4$ protocol. The first worker has access to port A, the second to port B, the third to port C, and the fourth to port D.

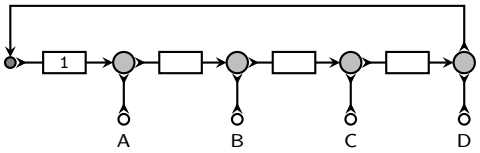


Figure 3.22: Circuit for a member of subfamily $\mathsf{Sequencer}_4$

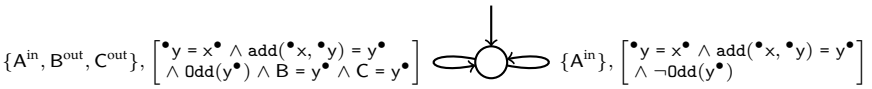
```
1  Sequencer(in[];) = {
2    FifoFull<'1'>(P1[1];P2[1])
3    mult { prod i:2..#in { Fifo(P1[i];P2[i]) } }
4      mult { prod i:1..#in { Replicator2(P2[i];P1[i+1],P3[i]) } }
5        mult { prod i:1..#in { SyncDrain(in[i],P3[i];) } }
6      mult Sync(P1[#in+1];P1[1])
7  }

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:

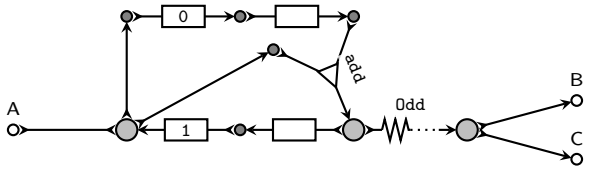


Figure 3.26: Circuit for a member of subfamily OddFibonacci$_2$

```
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

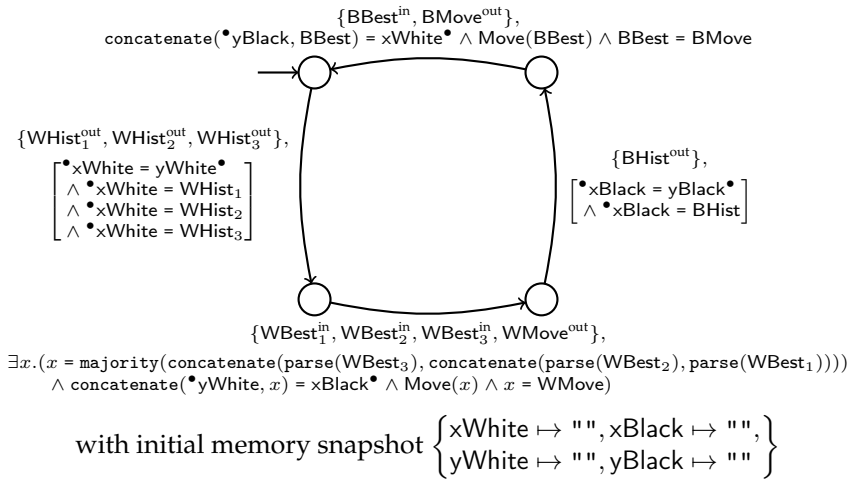


Figure 3.28: Constraint automaton for the Chess_3 protocol. For $i \in \{1, 2, 3\}$, every chess engine i for Team White has access to port WHist_i and to port WBest_i, the chess engine for Team Black has access to port BHist and port BBest, the display has access to ports WMove and BMove, and the chess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

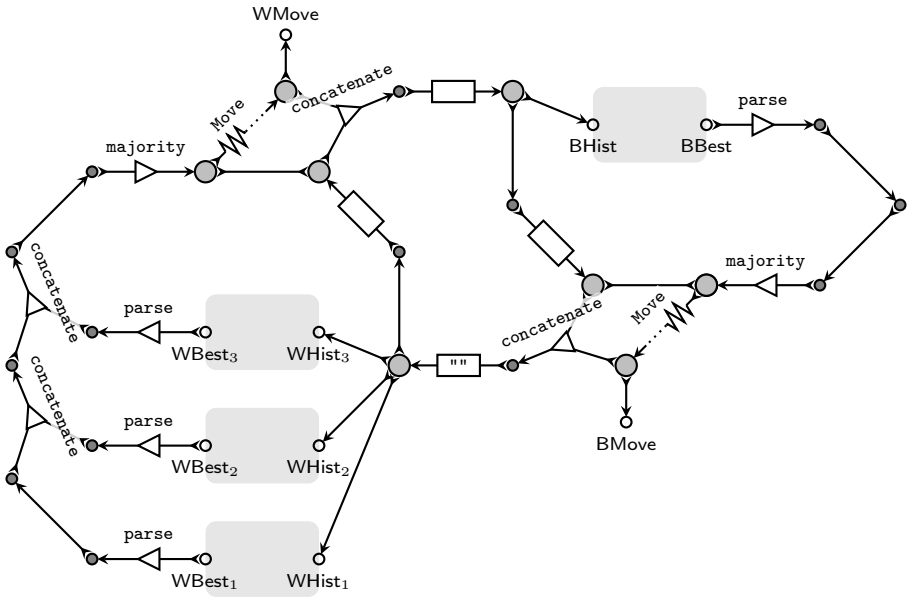


Figure 3.29: Circuit for a member of subfamily Chess_3, where gray boxes represent chess engines

```
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)

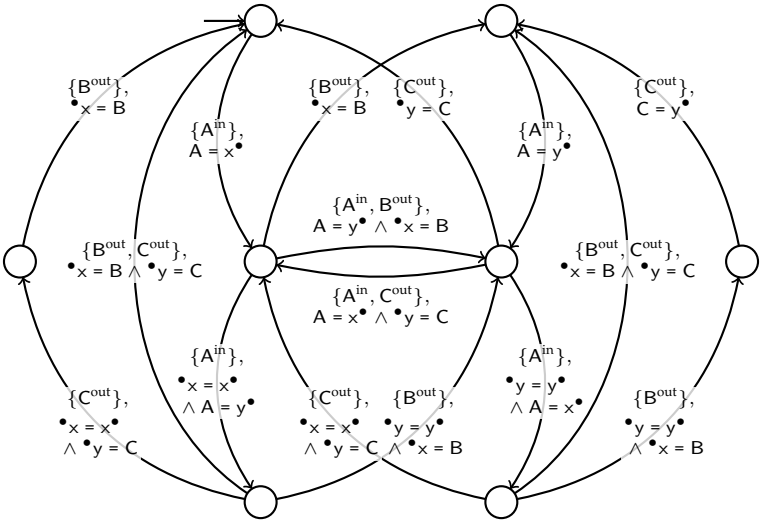
base, I pushed the information about interaction between slaves to the FOCAML implementation of the corresponding protocol specification, as demonstrated shortly. `PortBasedRelayRaceSlave`, then, illustrates a key point of using a separate DSL for interaction: worker subprograms in a complementary GPL 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.

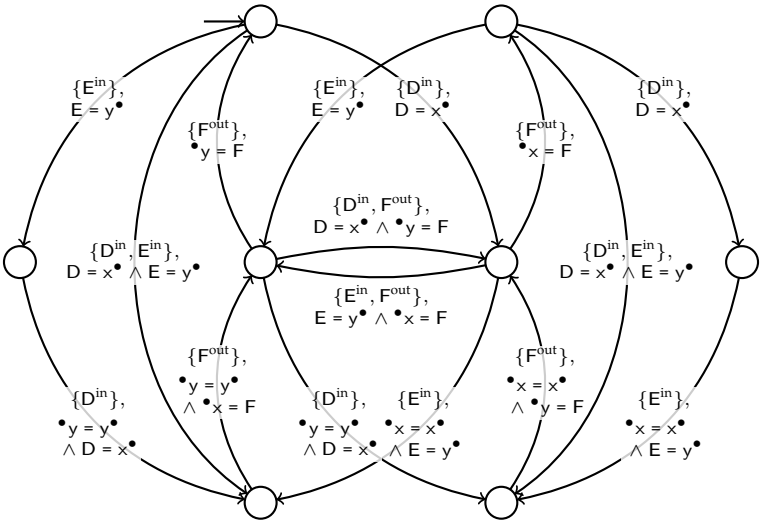
Members of MasterSlavesInteractionPatternA enforce an order in which the master signals its slaves and vice versa, achieved through members of Sequencer. The Java code for the master–slaves interaction pattern in Figure 3.31 also enforces such an order, through Java's fixed iteration order in loops. 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.

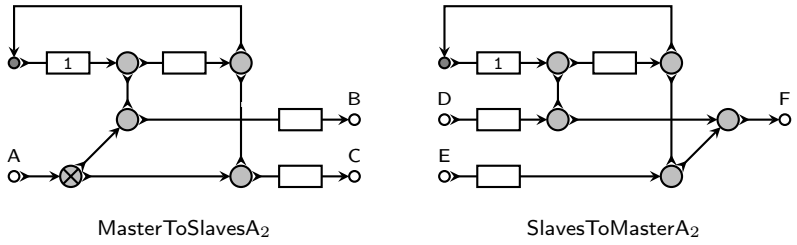


Figure 3.36: Circuits for members of subfamilies MasterToSlavesA$_2$ and SlavesToMasterA$_2$

```
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$_2$

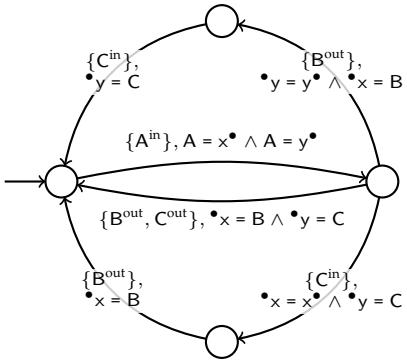


Figure 3.38: Constraint automaton for the MasterToSlavesB$_2$ protocol. The master has access to port A, one slave has access to port B, the other slave has access to port C, and the master and the slaves use buffers x and y for temporary storage of data.

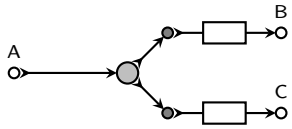


Figure 3.39: Circuit for a member of subfamily MasterToSlavesB$_2$

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-

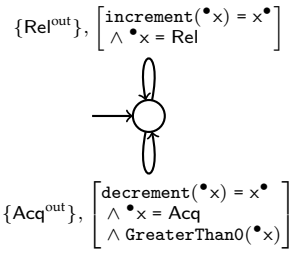


Figure 3.41: Constraint automaton for the Semaphore protocol.
One slave has access to port Acq, the other slave has access to port Rel, and the slaves use buffer x for temporary storage of data.

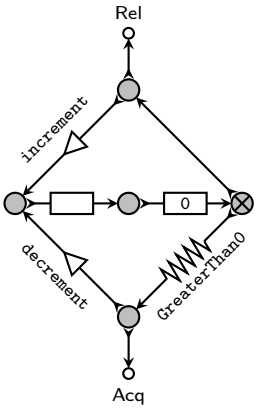


Figure 3.42: Circuit for a member of family Semaphore

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 $x > 0$.

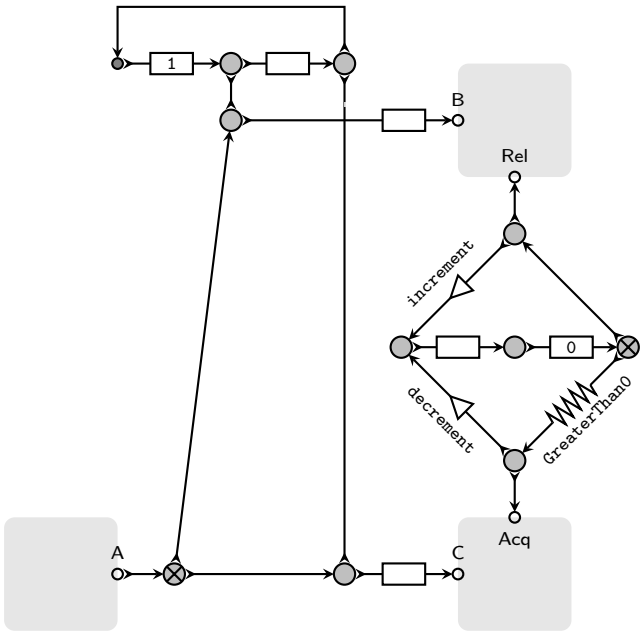


Figure 3.44: Circuit for the multiplication of a member of subfamily MasterToSlavesA_2 and a member of family Semaphore, where gray boxes represent the master (left) and its two workers (right)