Tokens? What tokens?

A gentle introduction to dataflow programming

Jörn W. Janneck

document edition 1

ASTG Technical Memo
Programming Solutions Group
Xilinx
August 5, 2007

Contents

Chapter 1

Introduction

This document is intended to provide a first introduction to dataflow programming using the CAL actor language and the NL network language. Rather than being exhaustive, we will try to present a number of fundamental and commonly-used constructs to help along with the first steps toward writing actor-based systems.

People may choose a dataflow specification for a number of reasons. One of the most common is that a dataflow description is the most appropriate way of expressing the algorithm, which is often the case in application domains such as digital signal processing or video and image processing. A good guide for whether this might be the case for a given problem is whether a description of the computation itself (as opposed to, say, the class structure or the use cases) starts with a diagram of blocks connected by arcs that denote the transmission of packets of information. If it is, chances are that this translates well into a dataflow program.

Another common reason for choosing dataflow is that the goal is an efficient parallel implementation which would be difficult or impossible to achieve using a sequential programming language. Sequential languages are notoriously difficult to parallelize in general, so efficient parallel implementations will usually require significant guidance from the user. A dataflow program provides simple, understandable, and powerful abstractions that allow the specification of as much or as little parallelism as is required, enabling tools to produce sophisticated implementations that exploit the concurrent structure of a computation.

When programming in dataflow, the programmer is typically constructing a concurrent description of a computational system, which is different from a common sequential program. Rather than being concerned with the step-by-step execution of an algorithm, a dataflow programmer builds a system of asynchronously communicating entities called *actors*. Much of the programming effort is directed toward finding a good factoring of the problem into actors, and toward engineering appropriate communication patterns among those actors.

This introduction covers the basic constructs required to build dataflow programs.

Chapter 2

The structure of actors

Actors perform their computation in a sequence of steps we call *firings*. In each of those steps...

1. the actor may consume tokens from its input ports,
2. it may modify its internal state,
3. it may produce tokens at its output ports.

Consequently, describing an actor involves describing its interface to the outside, the ports, the structure of its internal state, as well as the steps it can perform, what these steps do (in terms of token production and consumption, and the update of the actor state), and how to pick the step that the actor will perform next. This chapter discusses some of the constructs in the CAL language that deal with these issues.

2.1 Simple actors

One of the simplest actors that does anything at all is one that only copies a token from its input port to its output port. That is what the actor `ID` does:

```
actor ID () In ==> Out :
  action In: [a] ==> Out: [a] end
end
```

The first line declares the actor name, followed by a list of parameters (which is empty, in this case), and the declaration of the input and output ports. The input ports are those in front of the `==>` sign (here only one port named `In`), the output ports are those after it (in this case only one port named `Out`).[1]

[1]The CAL language is *optionally typed*, i.e. it allows type declarations to be present but it does not mandate them. However, tools such as compilers may require type declarations to be present in order to perform their task. In the interest of brevity and clarity, types declarations are omitted from this and the subsequent examples.

The second line defines an *action*. Actions are the beef of an actor—they describe the things that happen during a step that an actor takes. In fact, it is accurate to say that a step consists of executing an action. In general, actors may have any number of actions, but `ID` has only one.

Recall that when an actor takes a step, it may consume input tokens and produce output tokens. The action in `ID` demonstrates how to specify token consumption and production. The part in front of the `==>`, which we call *input patterns*, again pertains to input ports, and it specifies how many tokens to consume from which ports and what to call those tokens in the rest of the action. There is one input pattern in this action, `In: [a]`. It says that one token is to be read (and consumed) from input port `In`, and that the token is to be called `a` in the rest of the action. Such an input pattern also defines a condition that must be met for this action to fire—if the required token is not present, this action will not be executed. Therefore, input patterns do the following:

- They define the number of tokens (for each port) that will be consumed when the action is executed (fired).
- They declare the variable symbols by which tokens consumed by an action firing will be referred to within the action.
- They define a *firing condition* for the action, i.e. a condition that must be met for the action to be able to fire.[2]

The output side of an action is a little simpler—following the `==>` sign, the *output expressions* simply define the number and values of the output tokens that will be produced on each output port by each firing of the action. In this case, `Out: [a]` says, that exactly one token will be generated at output port `Out`, and its value is `a`.

It is worth noting that although syntactically the use of `a` in the input pattern `In: [a]` looks the same as the one in the output expression `Out: [a]`, their meanings are very different. In the input pattern, the name `a` is *declared*, it is introduced as the name of the token that is consumed whenever the action is fired. By contrast, the occurrence of `a` in the output expression *uses* that name.

It is permissible to omit the explicit naming of the port that an input pattern or output expression applies to if an action provides as many input patterns as there are input ports, or output expressions as there are output ports. In such a case, the patterns or expressions are matched by position against the port declarations. For instance, the following versions of `ID` are all equivalent to the original one above:

```
actor ID () In ==> Out :
  action In: [a] ==> [a] end
end
```

[2] There are a number of those conditions which we will encounter in the course of this chapter. They are summarized in section 2.7.

```
actor ID () In ==> Out :
  action [a] ==> Out: [a] end
end

actor ID () In ==> Out :
  action [a] ==> [a] end
end
```

The next example, `Add`, shows an actor that has two input ports. Like `ID`, it also has a single action, but this time, the action reads one token from each of the input ports. The single output token produced by this action is the sum of the two input tokens:

```
actor Add () Input1, Input2 ==> Output:
  action Input1: [a], Input2: [b] ==> Output: [a + b] end
end
```

Incidentally, this illustrates the difference between input patterns and output expressions—an expression such as `a + b` is a perfectly valid way of specifying the value of an output token inside an output expression, but it would be illegal in an input pattern.

Just as in the case of `ID`, we can write `Add` a little more concisely by omitting the ports in the description of the action:

```
actor Add () Input1, Input2 ==> Output:
  action [a], [b] ==> [a + b] end
end
```

One way of thinking about an actor is as an operator on *streams* of data[3]—sequences of tokens enter it on its input ports, and sequences of tokens leave it on its output ports. When discussing the operation of an actor, it is often useful to look at it as an operator on streams. For instance, say we look at the `Add` actor, at a point in time when the tokens 5, 7, -3 are on its `Input1` and 11, 7, and 0 are on its `Input2`, with no token so far produced at its `Output`. We could write this as

```
Input1:  [5, 7, -3], Input2:  [11, 7, 0] ==> Output:  []
```

or more concisely as

```
[5, 7, -3], [11, 7, 0] ==> []
```

if the order of ports is understood, in the same way in which we elide the input patterns and output expressions.

Now we can look at a *run* of the `Add` actor by looking at how the sequences of tokens evolve as the actor makes its steps. Starting from the sequences above, this would look as follows:

[3] I avoid here the term *function* or *stream function* because, as we will see, actors are not always functions in the mathematical sense of computing a unique value for each of their inputs.

```
    [5, 7, -3], [11, 7, 0] ==> []
--> [7, -3], [7, 0] ==> [16]
--> [-3], [0] ==> [16, 14]
--> [], [] ==> [16, 14, -3]
```

Note how inputs are consumed from the front of the input sequences, and outputs are produced (and appended to them) at their end.

During a firing, actors can consume more than one token from any input port, and the can produce more than one output token. The following actor `AddSeq` consumes two tokens from its single input port and adds them:

```
actor AddSeq () Input ==> Output:
  action [a, b] ==> [a + b] end
end
```

A run of `AddSeq` could look like this:

```
    [1, 2, 3, 4, 5, 6] ==> []
--> [3, 4, 5, 6] ==> [3]
--> [5, 6] ==> [3, 7]
--> [] ==> [3, 7, 11]
```

The actor `AddSub` produces two output tokens—one the sum, the other the difference between its input tokens:

```
actor AddSub () Input1, Input2 ==> Output:
  action [a], [b] ==> [a + b, a - b] end
end
```

This might be a run of this actor:

```
    [1, 2], [3, 4] ==> []
--> [2], [4] ==> [4, -2]
--> [], [] ==> [4, -2, 6, -2]
```

Actors can have parameters. They act as constants during the actor execution, and are given a concrete value when an actor is *instantiated* as part of an actor network—see chapter 4. The main purpose of actor parameters is to allow programmers to specify families of related actors, without having to duplicate a lot of code.

```
actor Scale (k) Input ==> Output:
  action [a] ==> [k * a] end
end
```

An instance of this actor with `k=7` could have this run:

```
    [3, 5, 8] ==> []
--> [5, 8] ==> [21]
--> [8] ==> [21, 35]
--> [] ==> [21, 35, 56]
```

2.2 Nondeterminism

Up to this point, all actors had a single action, although it was already mentioned that this need not be the case in general. Actors can have any number of actions, including none at all. The following actor, `NDMerge`, has two:

```
actor NDMerge () Input1, Input2 ==> Output:
  action Input1: [x] ==> [x] end
  action Input2: [x] ==> [x] end
end
```

The first action consumes a token from `Input1` and sends it to the output, the second does the same for `Input2`. Each for itself is very similar to the action in `ID`, in that they copy a token from an input port to an output port. However, both action copy tokens from different input ports to *the same* output port—and therein lies the rub.

To illustrate the problem, let us look at runs of this actor. This one is obvious:

```
    [1, 2, 3], [] ==> []
--> [2, 3], [] ==> [1]
--> [3], [] ==> [1, 2]
--> [], [] ==> [1, 2, 3]
```

And so is this one:

```
    [], [1, 2, 3]==> []
--> [], [2, 3] ==> [1]
--> [], [3] ==> [1, 2]
--> [], [] ==> [1, 2, 3]
```

But what happens if there are tokens available at both input ports?

```
    [1, 2], [3, 4] ==> []
--> ???
```

The issue here is that both actions have enough input tokens to fire, and the output will look different depending on which we choose. If we pick the first, we get

```
    [1, 2], [3, 4] ==> []
--> [2], [3, 4] ==> [1]
```

However, if we pick the second, we get

```
    [1, 2], [3, 4] ==> []
--> [1, 2], [4] ==> [3]
```

Clearly, it does make a difference which action is chosen, so the question is: What is the rule for determining which action gets to fire in such a case?

The answer is that there is no such rule. If more than one action satisfies all its firing conditions at any point in time, then the next action to fire is one of those actions, but the choice among them is not part of the actor specification. What this means is that the author of the actor has left this choice open, and that an implementation, or simulation, is free to pick whichever it deems best.[4]

nondeterminism

What we see here is called *nondeterminism*—a nondeterministic actor is one that, for the same input sequences, allows more than one run and more than one possible output.[5] Nondeterminism can be very powerful when used appropriately, but it can also be a very troublesome source of errors. A particular concern is that nondeterminism might be introduced into an actor inadvertently, i.e. the author thinks the actor is deterministic even though it isn't. One of the key design goals of the CAL language was to allow the description of nondeterministic actors, while at the same time permitting tools to identify possible sources of nondeterminism, so that they can warn the user about them.

A key consequence of a nondeterministic actor like `NDMerge` is that during an actual execution, its output may depend on the *timing* of its input. If both its input queues are empty, and `NDMerge` is waiting for input, then whatever input the next token arrives at may be the one that is copied next to the output.[6] Consequently, the scheduling of activities in the actor network, or the relative speeds of the actors feeding into an actor like `NDMerge` may affect the output of the system. This may, occasionally, by desirable, and at other times it may not. In any event, it is a property that one needs to be aware of.

One way to look at nondeterminism of the kind that makes an actor dependent on the precise timing of token arrivals is that such an actor only *appears* to be nondeterministic if we look at it as an operator on streams, because that view abstracts from the temporal properties of the execution, and thus purposefully removes information that is used to determine the sequence in which actions fire. From the perspective of the CAL*language*, this is not entirely accu-

[4] Note that this does **not** imply any kind of randomness or any sort of probability distribution among actions, nor is it required that an implementation be in any way "fair" in implementing whatever mechanism ends up making that choice. In other words, it is perfectly fine to create an entirely deterministic implementation, one, e.g., that always favors one of the firable actions over the others. The point is, we don't know, and—assuming we wrote the actor deliberately that way—we do not care, either.

[5] To be very precise, we would have to distinguish between an actor having more than one run for a given input, and having more than one output, since it is possible that two different runs result in the same output. Thus, technically, an actor that has action-level nondeterminism, in the sense that it would allow more than one action to fire at any point during its execution, could still look entirely deterministic from the outside if all runs for the same input result in the same output sequences. In the following we will gloss over that distinction, as it is not likely to be a very relevant one in practice.

[6] Again, it depends on an implementation whether that is actually the case. However, it **may** be the case, which is the point here.

rate,[7] but even so, it is easy to write nondeterministic actors that would not be deterministic even if we knew everything about the timing of the tokens and the actor implementation[8]—such as the following:

```
actor NDSplit () Input ==> Output1, Output2:
  action [x] ==> Output1: [x] end
  action [x] ==> Output2: [x] end
end
```

Admittedly, it may not immediately be obvious what this actor could be used for, but it is an illustration of the nature of nondeterminism in dataflow.

2.3 Guarded actions

So far, the only firing condition for actions was that there be sufficiently many tokens for them to consume, as specified in their input patterns. However, in many cases we want to specify additional criteria that need to be satisfied for an action to fire—conditions, for instance, that depend on the values of the tokens, or the state of the actor, or both. These conditions can be specified using *guards*, as for example in the `Split` actor:

```
actor Split () Input ==> P, N:
  action [a] ==> P: [a]
  guard a >= 0 end

  action [a] ==> N: [a]
  guard a < 0 end
end
```

The `guard` clause of an action contains a number of expressions that all need to be `true` in order for the action to be firable. For the first action to be firable, the incoming token needs to be greater or equal to zero, in which case it will be sent to output `P`. Otherwise that action cannot fire. Conversely, for the second action to be firable, the token needs to be less than zero, in which case it is sent to output `N`. A run of this actor might look like this:

```
    [1, -2, 0, 4] ==> [], []
--> [-2, 0, 4] ==> [1], []
--> [0, 4] ==> [1], [-2]
```

[7] The reason is that CAL does not provide any guarantees about the "reaction time" of an actor. For instance, in the case of `NDMerge`, we need to assume that the actor is waiting on both input queues and ready to fire on te first token that arrives before another token appears on the other input port. Nothing in the language requires that this is so, and it may well not be in some implementation.

[8] One of the key benefits that accrues from using dataflow and actors for the construction of concurrent systems is the abstraction this methodology affords from concrete physical timing. It is usually a bad idea to break that abstraction by using information about implementation behavior—even if that information turns out to be correct, and the resulting system happens to work.

```
--> [4] ==> [1, 0], [-2]
--> [] ==> [1, 0, 4], [-2]
```

There are three things of note about this actor. First, the way it is written, the guard conditions happen to be *exhaustive*—i.e. the guard conditions cover all possible inputs—assuming only real numbers (or integers) are seen at the input port, there will never be an input such that neither of the two guards is true. For instance, say we modified the first guard just slightly:

```
actor SplitDead () Input ==> P, N:
  action [a] ==> P: [a]
  guard a > 0 end

  action [a] ==> N: [a]
  guard a < 0 end
end
```

This actor will run into trouble if it ever encounters a zero token, because none of its actions will be able to fire on it. As a consequence, that token will never be consumed, and the actor will no longer be able to fire at all—it will be *dead*.

```
    [1, -2, 0, 4] ==> [], []
--> [-2, 0, 4] ==> [1], []
--> [0, 4] ==> [1], [-2]
```

It's not illegal to write actors that terminate on some input, and in fact it may be important to have a few of those in some system. But it is a pitfall that one needs to be aware of.

Secondly, the guard conditions are also *disjoint* in addition to being exhaustive—i.e., none of the two guards are true at the same time. Modifying the second guard of `Split` a little, we get this actor:

```
actor SplitND () Input ==> P, N:
  action [a] ==> P: [a]
  guard a >= 0 end

  action [a] ==> N: [a]
  guard a <= 0 end
end
```

Even though `SplitND` has only guarded actions, it is still nondeterministic, because for some input (zero), both actions can fire. In other words, in addition to the runs of `Split`, this actor also has, e.g., this run:

```
    [1, -2, 0, 4] ==> [], []
--> [-2, 0, 4] ==> [1], []
--> [0, 4] ==> [1], [-2]
--> [4] ==> [1], [-2, 0]
--> [] ==> [1, 4], [-2, 0]
```

Finally, note that guard conditions can "peek" at the incoming tokens without actually consuming them—if the guards happen to be `false` or the action is not fired for some other reason, and if the token is not consumed by another action, then it remains where it is, and will be available for the next firing. (Or it will remain there forever, as in the case of the zero token in front of `SplitDead`, which is never removed because the actor is dead.)

The `Select` actor below is another example of the use of guarded actions. It is similar to the `NDMerge` actor in the sense that it merges two streams (the ones arriving at its `A` and `B` input ports). However, it does so according to the (Boolean) values of the tokens arriving at its `S` input port.

```
actor Select () S, A, B ==> Output:
  action S: [sel], A: [v] ==> [v]
  guard sel end

  action S: [sel], B: [v] ==> [v]
  guard not sel end
end
```

2.4 Actors with state

In all the actors so far, nothing an action firing did would in any way affect subsequent firings of actions of the same actor. Using *state variables*, action firings can leave information behind for subsequent firings of either the same or a different action of the same actor.

A simple example of this is the `Sum` actor:

```
actor Sum () Input ==> Output:
  sum := 0;

  action [a] ==> [sum]
  do
    sum := sum + a;
  end
end
```

This actor maintains a variable in which it accumulates the sum of all tokens it has seen (and consumed). The declaration `sum := 0;` introduces the variable name and also assigns the variable an initial value. The action, in addition to consuming an input token and producing an output token, now also *modifies* the actor state by assigning a new value to the state variable. The next time this actor fires, the state variable will have that new, updated, value. An run of `Sum` might be this:

```
    [1, 2, 3, 4] ==> []
--> [2, 3, 4] ==> [1]
```

```
--> [3, 4] ==> [1, 3]
--> [4] ==> [1, 3, 6]
--> [] ==> [1, 3, 6, 10]
```

Note that the value that is produced by the output expression is the value of the state variable *at the end of the action firing*, i.e. after the variable has been updated. This is a general rule, and important to keep in mind: If state variables occur in output expressions, the value that they refer to is the value at the end of the action firing. If the action modified that state variable, then it is the new value that will be used.

Sometimes, one would like to use the old value, the one that was valid at the beginning of the action firing, before any possible updates might have happened. The `old` keyword can be used to identify that value. It may only be used with state variables:

```
actor SumOld () Input ==> Output:
  sum := 0;

  action [a] ==> [old sum]
  do
    sum := sum + a;
  end
end
```

This actor would have the following run:

```
    [1, 2, 3, 4] ==> []
--> [2, 3, 4] ==> [0]
--> [3, 4] ==> [0, 1]
--> [4] ==> [0, 1, 3]
--> [] ==> [0, 1, 3, 6]
```

Sometimes, state is used to control the selection of actions. Recall the `Select` actor from the previous section:

```
actor Select () S, A, B ==> Output:
  action S: [sel], A: [v] ==> [v]
  guard sel end

  action S: [sel], B: [v] ==> [v]
  guard not sel end
end
```

The way this actor is written, the selection of the next input token and the actual copying of the token to the output is one atomic step. Suppose we want to rewrite that actor to perform these two things in two distinct actions. The actor would then execute in two stages—in the first, it would wait for a token on input `S`. Once it read that token it would, depending on its value wait for a

data token on either A or B. Once that arrived, it would copy it to the output, and go back to waiting for a token on S.

The following actor IterSelect is written in that way. Its state variable state is used to select the action that is waiting for input, depending on whether the variable is 0, 1, or 2. Initially, by making 0 the initial value of state, IterSelect waits for input on S, and then it proceeds as described above.

```
actor IterSelect () S, A, B ==> Output:
  state := 0;

  action S: [sel] ==>
  guard state = 0 do
    if sel then
      state := 1;
    else
      state := 2;
    end
  end

  action A: [v] ==> [v]
  guard state = 1 do
    state := 0;
  end

  action B: [v] ==> [v]
  guard state = 2 do
    state := 0;
  end
end
```

Isn't this incorrect? Since the S token can only be consumed when state=0, and

Note that Select and IterSelect are almost, but not entirely, equivalent. First of all, IterSelect makes twice as many steps in order to process the same number of tokens. Secondly, it actually reads, and therefore consumes, the S input token, irrespective of whether a matching data token is available on A or B.

Unlike the previous examples, this actor uses guards that depend on an actor state variable rather than on an input token. Of course, combinations are possible, as in this example:

```
actor AddOrSub () Input ==> Output :
  sum := 0;

  action [a] ==> [sum]
  guard a > sum
  do
    sum := sum + a;
```

```
  end

  action [a] ==> [sum]
  guard a <= sum
  do
    sum := sum - a;
  end
end
```

This actor would have a run such as this:

```
    [1, 2, 3, 4] ==> []
--> [2, 3, 4] ==> [1]
--> [3, 4] ==> [1, 3]
--> [4] ==> [1, 3, 0]
--> [] ==> [1, 3, 0, 4]
```

2.5 Schedules

The `IterSelect` actor of the previous section illustrated the use of state to control the selection of actions. This is an extremely common thing to do in practice, and the CAL language provides special syntax for this purpose in the form of *schedules*. Conceptually, one can think of schedules as codifying a particular pattern of using a state variable—they do not add anything to the language in terms of expressiveness. The rationale for using schedules is twofold: (1) They are usually easier to use and less error prone than using a state variable and lots of guards and assignments. (2) Tools can use the information encoded in a schedule more easily, and thus recognize regularities in the actor that might help them to produce more efficient code, or perform other analyses that help in implementation and design.

A version of `IterSelect` using a schedule looks like this:

```
actor IterSelect () S, A, B ==> Output:
  readT: action S: [s] ==>
         guard s end

  readF: action S: [s] ==>
         guard not s end

  copyA: action A: [v] ==> [v] end

  copyB: action B: [v] ==> [v] end

  schedule fsm init:
    init (readT) --> waitA;
    init (readF) --> waitB;
```

```
    waitA (copyA) --> init;
    waitB (copyB) --> init;
  end
end
```

First, let us look at the labels in front of the actions—`readT`, `readF`, `copyA`, and `copyB`. These are *action tags*, and are used to identify actions further down in the schedule.[9]

Then there is the schedule itself. Basically, it is a textual representation of a finite state machine, given as a list of possible state transitions. The states of that finite state machine are the first and the last identifiers in those transitions—in this case, `init`, `waitA`, `waitB`. Relating this back to the original version of `IterSelect`, these states are the possible *values* of the `state` variable, i.e. 0, 1, and 2. The initial state of the schedule is the one following `schedule fsm`—in this case, it is `init`.

Each state transition consists of three parts: the original state, a list of action tags, and the following state. For instance, in the transition
`init (readT) --> waitA;`
we have `init` as the original state, `readT` as the action tag, and `waitA` as the following state. The way to read this is that if the schedule is in state `init` and an action tagged with `readT` occurs, the schedule will subsequently be in state `waitA`.

One thing worth noting is that the number of actions has increased—instead of the original three, the new version with the schedule now has four actions. The reason is that an action can no longer directly assign the successor state, as it did in the original, where depending on the value of the token read `state` would be assigned either the value 1 or 2. In the version with a schedule, that state modification is implicit in the structure of the state machine, and it happens depending on which action fires. Accordingly, the condition that checks the value of the token has moved from within the body of the action to the guards of the two actions tagged `readT` and `readF`.

Let us do this again with a slightly smaller example, another actor that merges two streams.[10] Suppose we want to make sure that merging happens more deterministically than it did in `NDMerge`, i.e. we alternate between reading from the two inputs. That is what `AlmostFairMerge` does—it is not perfectly fair, as it is biased with respect to which input it starts reading from. But once it is running, it will strictly alternate between the two:

```
actor AlmostFairMerge () Input1, Input2 ==> Output:
  s := 0;

  action Input1: [x] ==> [x]
```

[9] Action tags are **not** action *names*, in the sense that they need not identify actions uniquely—the same tag might be used with more than one action.

[10] Merge actors are somewhat of the Dining Philosopher's Problem of dataflow programming: small and relatively easily understood, they can be used to illustrate a variety of issues and constructs in dataflow and tools and languages dealing with dataflow.

```
  guard s = 0
  do
    s := 1;
  end

  action Input2: [x] ==> [x]
  guard s = 1
  do
    s := 0;
  end
end
```

Obviously, this actor has two states, depending on which port it is waiting for input. A simple schedule can be used to express this logic much more succinctly:

```
actor AlmostFairMerge () Input1, Input2 ==> Output:

  A: action Input1: [x] ==> [x] end

  B: action Input2: [x] ==> [x] end

  schedule fsm s1:
    s1 (A) --> s2;
    s2 (B) --> s1;
  end
end
```

2.6 Priorities

Consider the following actor:

```
actor ProcessStreamND () In, Config ==> Out:
  c := initialConfig();

  action Config: [newC] ==>
  do
    c := newC;
  end

  action In: [data] ==> [compute(data, c)] end
end
```

This actor is clearly nondeterministic. As long as it has only input on one of its input ports, everthing is unambiguous. But, just like `NDMerge`, as soon as input is available on both input ports, it could fire either of its two actions, and

there is nothing in that actor specification which would predispose it to choose one over the other.

Suppose now that this actor processes, e.g., audio data that continuously streams in on its `In` input port, and that this processing depends on the value of its state variable `c`—imagine `c` containing the setting of the volume dial. Every now and then, the user turns that dial, and a new value for `c` is sent to this actor. Clearly, it is not irrelevant in which order the two actions fire. In fact, we would like to make sure that the first action fires as soon as possible, so that the new user setting will take effect. More precisely, we would like to express the requirement that, should both actions be able to fire, the first one will be fired next.

Interestingly, none of the language constructs so far would allow us to do this. Unlike in this case of schedules, which could be regarded syntactic sugar because they could be reduced to existing elements of the language (state variables, guards, and assignments), this situation does in fact require a true extension—action *priorities*.

The basic idea is to add a number of inequalities that relate actions with respect to their firing precedence.[11] In our example, this leads to the following solution:

```
actor ProcessStream () In, Config ==> Out:
  c := initialConfig();

  config:  action Config: [newC] ==>
           do
             c := newC;
           end

  process: action In: [data] ==> [compute(data, c)] end

  priority
    config > process;
  end
end
```

Just as in the case of schedules, we use action tags to identify actions that we want to refer to later on—this time within the priority inequality. The priority block contains only one such inequality, relating the action tagged `config` to the one tagged `process`, giving the former priority over the latter.

Of course, even this version is still very much timing-dependent. In this case, that need not be a problem, and in fact is probably a requirement for this actor to perform its function. But in general, it is important to understand that

[11] Technically, the *transitive closure* of these inequalities defines a *partial order* among the actions. This means that if we have inequalities $A > B$ and $B > C$, we infer that by transitivity, $A > C$. If there is an action D, which is not either directly or transitively related to A, B, or C, then there is no order specified between them and D.

priorities, especially when used as in the previous example, need to be well-understood to yield the correct results. Especially when information about the timing of the communication within the network is vague, it is probably best to think of them as strong implementation directives.

IMPLEMENTATION NOTE.
Note that a correct implementation of priorities implies something like a "transactional" model for action selection. The idea is that the firability of all actions is evaluated in a single, defined state, which includes the state of the input queues. Consider the following example:

```
actor PriorityTest () In1, In2 ==> Out :
  A: In1: [a] ==> [1] end
  B: In2: [a] ==> [2] end
  C: In1: [a] ==> [3] end

  priority
    A > B;
    B > C;
  end
end
```

The actions labeled `A` and `C` both only require one token on input `In1`, but `A` has a higher priority. Therefore, it is impossible to ever see a 3 as an output token.
However, a naïve implementation might go down the list of actions, first testing the firability of the first action. Say no token is present at `In1` at that point, so the first action is not firable, and testing proceeds with the second action. Suppose again that no token is present at `In2` either, so testing proceeds with the third action. However, say a token has arrived meanwhile at `In1`. It would be incorrect for an implementation to proceed testing the third action and conclude that it can be fired. Instead, it has to take a "snapshot" of the inputs and test the firability of all actions under the same conditions.

2.6.1 A harmless use of priorities

In spite of the various ways in which the result of using priorities may be non-determinate, there is one kind of use that is fairly straightforward, easily identifiable, and it yields perfectly deterministic results.

Consider the following "routing" actor:

```
actor Route () A ==> X, Y, Z:
```

```
  action [v] ==> X: [v]
  guard P(v) end

  action [v] ==> Y: [v]
  guard Q(v) and not P(v) end

  action [v] ==> Z: [v]
  guard not Q(v) and not P(v) end
end
```

Assume that `P` and `Q` are two predefined Boolean functions on whatever tokens `Route` receives on its `A` input port. The job of this actor is to send all those tokens that satisfy `P` to output `X`, all of the remaining ones that satisfy `Q` to `Y`, and all other tokens to `Z`. In a sense, this is a dataflow version of a case statement.

In order for this to work properly, we have to ensure that the guards of al actions are (a) exhaustive and (b) disjoint, just like the guards of the `Split` actor in section 2.3. Now instead of the rather cumbersome and error-prone construction above, the same actor can be written more robustly and more concisely as follows:

```
actor Route () A ==> X, Y, Z:
  toX: action [v] ==> X: [v]
       guard P(v) end

  toY: action [v] ==> Y: [v]
       guard Q(v) end

  toZ: action [v] ==> Z: [v] end

  priority
      toX > toY > toZ;
  end
end
```

This actor is completely deterministic, it is shorter, and possibly easier to understand than the previous version. What is more, tools can easily analyze this actor to be deterministic—something which would be considerably more difficult for the previous version, because it would involve reasoning about the structure of the guards. All it takes to show that the second version is deterministic is a look at the input patterns and the priority structure.

2.7 Summary: action selection

An actor executes by alternating between two phases: 1. Selecting the next action to fire. 2. Executing/firing that action. Action selection itself proceeds as follows:

1. Determine the set of *eligible* actions. An action is elegible if (a) it does not have an action tag or (b) its action tag is among those that label an arc leading away from the current state of the schedule. In other words, eligible actions are those that can possibly be fired according to the schedule alone. If there is no schedule, then all actions are eligible.
2. From those eligible actions, determine the subset of *activated* actions. An action is activated iff it is eligible and the following conditions are fulfilled: (a) There are enough tokens available on the input ports to satisfy the input patterns. (b) All the guard expressions evaluate to `true`.
3. From the activated actions, determine the subset of *firable* actions. An action is firable iff it is activated and there is no other activated action with a higher priority.

Any firable action may be selected—there is no policy prescribed for picking the next action to execute among the firable actions.

Chapter 3

Statements and expressions in CAL

The previous chapter focused primarily on those constructs in CAL that are related to actor-specific concepts—token input and output, actions, controlling the action selection and so forth. This chapter discusses the more "pedestrian" parts of CAL, the statements and expressions used to manipulate data objects and express (sequential) algorithms. This part of the language is similar to what can be found in many procedural programming languages (such as C, Pascal, Java, Ada, ...), so we will focus on areas that might be slightly different in CAL.

3.1 Expressions

Unlike languages such as C, CAL makes a strong distinction between statements and expressions. They have very distinct roles, very distinct meanings, and they can never be used interchangeably. An *expression* in CAL is a piece of code whose sole purpose is to compute a *value*. We also say that an expression *has* a value, or that it *evaluates* to a value. For most expressions, the value that they evaluate to will depend on the values of one or more variables at the time when the expression is evaluated. Since variable values may change over time, the same expression may have different values when evaluated at different points in time.

3.1.1 Atomic expressions

constants

Probably the most fundamental expressions are *constants*. These are expressions whose values are guaranteed not to depend on any variables. Constants in CAL are the Boolean values `true` and `false`, numerical constants such as `11`, `-1`, `3.14`, and `1.3806503e-23`, and strings enclosed in quotation marks like `"abc"`, `"another string"` and `""`, as well as the null value `null`.

Another group of basic expressions are *variable references*. Syntactically, a variable is any sequence of letters, digits, and the "_" character that (a) does not start with a digit and (b) is not a keyword.[1] One important property of expressions is that they are guaranteed not to change variables (we also say they have no *side effects*)—consequently, within an expression, multiple references to the same variable will always yield the same result.[2]

variables

expressions do not affect variables

3.1.2 Simple composite expressions

CAL provides *operators* of two kinds to build expressions: unary and binary. A unary operator in CAL is always a *prefix* operator, i.e. it appears before its single operand. A binary operator occurs between its two operands. These are examples of expressions using unary operators: `-a`, `#s`. The unary `-` operator negates the value of its operand, which must be a number (i.e. it must *evaluate to* a number). The unary `#` operator applies to lists (and other collections), and computes their *size*, i.e. the number of elements in them. (More on lists in section 3.1.3.)

operators

These are examples of uses of binary operators: `a + 1`, `a + b + c`, and `a + b * c`. Of course, the usual rules of operator binding apply, so that the last expression can also be written `a + (b * c)`.

There is also a *conditional expression*, which works much like the ?:-operator in C-like languages, albeit with a slightly different syntax. For example, one can write

```
if a > b then 0 else 1 end
```

where `a > b` is the condition, and `0` and `1` are the expressions that are evaluated in case the condition is true or false, respectively. Note that the conditional expression is different from operators not only in the number of expressions it contains (three instead of one or two), but also in the way it evaluates those expressions. If the condition is true, then only the then-branch expression matters for the result of the conditional expression, and therefore it is guaranteed to be defined even if the else-branch expression, for instance, is not. For example,

```
if a = 0 then null else 1/a end
```

will produce a defined value (`null`) if `a` is zero, even though the else-branch expression is undefined in that case.

`if ... then ... else ... end`

3.1.3 Lists

Collections are composite data objects built from a number of other objects. A common example of a collection is a list, which can be constructed like this: `[1, 2, 3]` This builds a list of three elements, the integers `1`, `2`, and `3`. The expression `[]` results in the empty list. The elements in such a list expression

lists

[1] Cf. the CAL Language Report [1] for a complete list of keywords.

[2] The only exception to this rule are complex expressions that introduce local variables within the expression, and use them. In that case, it could be possible that the same variable symbol refers to the local variable in one part of the expression, and to a non-local variable is another.

may be arbitrary expressions:

```
[a, a + 1, a * a]
```

With, say, `a = 7`, this expression would evaluate to a list of three elements 7, 8, and 49.

Lists can be built from existing lists using a construction called *list comprehension*. It looks like this:

list comprehensions

```
[a * a :  for a in [1, 2, 3, 4]]
```

This results in a list with the elements 1, 4, 9, and 16. The expression in front of the colon, `a * a`, is an *element expression*. Because of the *generator* that follows the colon, it is evaluated for the variable `a` bound to each element of the *generator list*, in this case 1, 2, 3, and 4.

Comprehensions may contain more than one generator, as in this example:

```
[a * b :  for a in [2, 3, 5], for b in [7, 11]]
```

In this case, the result list is constructed by binding the variables `a` and `b` to all combinations of values from the respective generator lists. The further to the right a generator is, the faster does its generator variable vary over the elements of the generator list. In the example above, the `b` generator is to the right of the `a` generator. Consequently, after the first element, which is 2 * 7 = 14, the next element is obtained by taking the next element in the *econd* generator, yielding 2 * 11 = 22, rather than 3 * 7 = 21. Consequently, the list resulting from evaluating the comprehension above contains the elements 14, 22, 21, 33, 35, 55 in that order.

Similarly, a list comprehension can contain more than one element expression. For example,

```
[a, a * a :  for a in [2, 3, 5]]
```

results in a list containing 2, 4, 3, 9, 5, 25 in that order.

For more details on comprehensions, please refer to the CAL Language Report [1], section 6.10.

In order to extract a part of a collection such as a list, one needs to use an *indexer*. An indexer is an expression that contains (a) an expression computing a *composite object*, such as a list, and one or more expressions computing *indices*. The indices are identifying a *location* inside the composite object, at which the part of it that we want to use resides. In the case of lists, indices are the natural numbers from zero to the length of the list minus one. So for instance, if `b` is the list `[1, 1, 2, 3, 5, 8]`, then the indexer

indexer

```
b[4]
```

would evaluate to `5`. So does, by the way, the rather confusing-looking expression

```
[1, 1, 2, 3, 5, 8][4]
```

List miscellanea

The function `Integers` takes two arguments and computes a list of all integers between them, inclusively, and in order. For instance, `Integers(3, 7)` results in the list `[3, 4, 5, 6, 7]`. If the second argument is greater than the first, the resulting list is empty. The `..` operator serves as a short form of

`Integers(...)`

`a .. b`

the Integers function—the term `Integers(a, b)` is equivalent to `a .. b`.

The `#` operator is used to determine the *size* of a list, i.e. the number of elements in it. For instance, `#[1, 1, 2, 3, 5, 8]` evaluates to `6`. This can be used to to make sure that an index into a list is actually valid, and also to iterate over the elements of a list. For example, if `a` contains a list, then the following expression computes the reverse of that list:

operator

```
[a[#a-i] :  for i in 1..#a)]
```

Lists can be concatenated using the `+` operator, so for example, the expression

+ operator

```
[1, 2, 3] + [4, 5]
```

results in the list `[1, 2, 3, 4, 5]`. Concatenating a list with an empty list has no effect.

3.1.4 Functions

Functions encapsulate expressions and allow the programmer to parameterize them. For instance,

```
function double (x) : 2 * x end
```

Here, `double` is the *function name*, `x` is a *parameter*, and the expression between the colon and the `end` is the *function body*.

One thing to note about functions is that they contain exactly one expression in their body. Because assignments are *statements* (see section 3.2), no variables can be changed through the invocation of a function.

Functions may be recursive:

```
function fib (n) :
    if n < 2 then
        1
    else
        fib(n-1) + fib(n-2)
    end
end
```

Functions defined in the same scope may be mutually recursive:

```
function A (m, n) :
    if m <= 0 then
        n + 1
    else
        B(m - 1, n)
    end
end

function B (m, n) :
    if n <= 0 then
        A(m, 1)
```

```
    else
        A(m, A(m + 1, n - 1))
    end
end
```

The evaluation of expressions containing function applications, like the evaluation of expressions in general, can easily take advantage of some fine-grained parallelism inherent in CAL—for instance, in the expression `F(G(x), H(x, y))`, the order in which `G(x)` and `H(x, y)` are evaluated does not change the result, and in fact they may be evaluated in parallel. This is a consequence of the absence of side-effects for CAL expressions.

3.2 Statements

In some ways, statements in CAL are just the opposite of expressions: they do not have a "return value", but they can change the values of variables. Indeed, changing the values of variables is the whole point of statements. That is what they do.

Statements are executed in strict sequential order, and unless otherwise specified (see the section on control flow constructs, 3.2.2), the execution of statements proceeds in the order in which they appear in the program text, which means that any variable changes produced by a statement may affect the execution of subsequent statements.[3]

sequential execution

3.2.1 Assignments

So in the same way that an expression can be characterized by describing the value that it evaluates to, a statement can be described by how it changes variables. The most fundamental statement is an *assignment*, and the simplest assignment looks like these:

assignment

```
a := 0;
n := n + 1;
buf := [buf[i] :  for i in 1 .. #buf - 1)] + [a];
```

All of these simply change the old value of a variable to a new one.

Often variables contain composite objects, for instance a list of things rather than, say, an integer. In such a case, it is often desirable to change only a *part* of the object, while leaving the rest of it as before. This can be achieved using a simple assignment as above, e.g. like this:

```
m := [if i = k then v else m[i] end :
     for i in 0 .. #m - 1)];
```

[3] Of course, this strict sequentiality is the programmer's model of statement execution. An actual *implementation* of the program may execute statements differently, as long as the result is indentical to the one obtained from a purely sequential execution. In particular, an implementation may parallelize statements, if it can guarantee that this parallelization will not affect the outcome.

The right-hand side of this assignment computes a list that only differs from the list in `m` by one element: at position `k`, it has the value `v`. After assigning that list to `m`, the effect is the same as if we had *modified* the original value of `m` at *position* `k`. Clearly, that is a very roundabout way of achieving this, which is why there are *indexed assignments* to make this more concise. The assignment above is equivalent to the following indexed assignment:

indexed assignment

```
m[k] := v;
```

3.2.2 Control flow

As in most other programming languages, there are constructs to control the order in which the statements within a program are executed. The most basic one is the *conditional statement*:

conditional statement

```
if n = 0 then
    b := [];
else
    b := [n + i : for i in 1 .. n];
end
```

Unlike for conditional expressions, a conditional statement may omit the `else`-branch:

```
if val < 0 then
    val := 0;
end
```

Loops are another way of controlling the flow of execution. The simplest one is the `while`-loop, which executes a piece of code over and over again as long as a specified condition remains true:

`while`

```
sum := 0;
i := 0;
while i < #a do
    sum := sum + a[i];
    i := i + 1;
end
```

The above loop would iterate over the valid indices of the list in variable `a`, starting at `0` and continuing until it is no longer true that `i < #a`. The *body* of the loop adds the element `a[i]` to `sum`, and also increments the variable `i` itself.

Iterating over the elements of a collection is so common that there is a special construct for it, the `foreach`-loop. Using it, the loop above could also be written like this:

`foreach`

```
sum := 0;
foreach v in a do
    sum := sum + v;
end
```

The part of this loop that directly follows the `foreach` keyword is a generator, much like those in list comprehensions (see section 3.1.3). And like comprehensions, `foreach`-loops can have more than one generator:

```
sum := 0;
foreach x in X, foreach y in Y do
    sum := sum + (x * y);
end
```

3.2.3 Procedures

Procedures are used to abstract and parameterize sequences of statements, just as functions (section 3.1.4) are abstracting and parameterizing expressions. For instance,

```
procedure DP (X, Y) begin
    sum := 0;
    foreach x in X, foreach y in Y do
        sum := sum + (x * y);
    end
end
```

Such a procedure can be invoked in the usual way:

```
DP(M[i], N[j]);
if sum > 0 then
    ...
```

Chapter 4

Building systems from actors

In practice, dataflow systems consist of several actors connected by channels along which they pass tokens to each other. There are many ways in which those actor networks can conceivably be built—this section shows some aspects of a textual Network Language (NL) [2] that we use for this purpose.

4.1 Building simple networks

The two main things that need to be done when building a network of actors are

1. instantiating a number of actors (which includes passing parameter values to those instances), and
2. creating the connections between them.

In addition, a network of actors can itself have input and output ports, which may be connected to the ports of actors inside it.

Let us start with two simple actors:

```
actor InitialTokens (tokens) In ==> Out:
  A: action ==> [tokens] repeat #tokens end

  B: action [a] ==> [a] end

  schedule fsm s0:
   s0 (A) --> s1;
   s1 (B) --> s1;
  end
end

actor Add () A, B ==> Result:
```

```
  action [a], [b] ==> [a + b] end
end
```

Note how the `InitialTokens` actor is firing its action labeled `A` once at the beginning, producing all the tokens it has been given as a parameter containing a list of tokens.[1] From then on, it fires only the action labeled `B`, which copies its input token to its output.

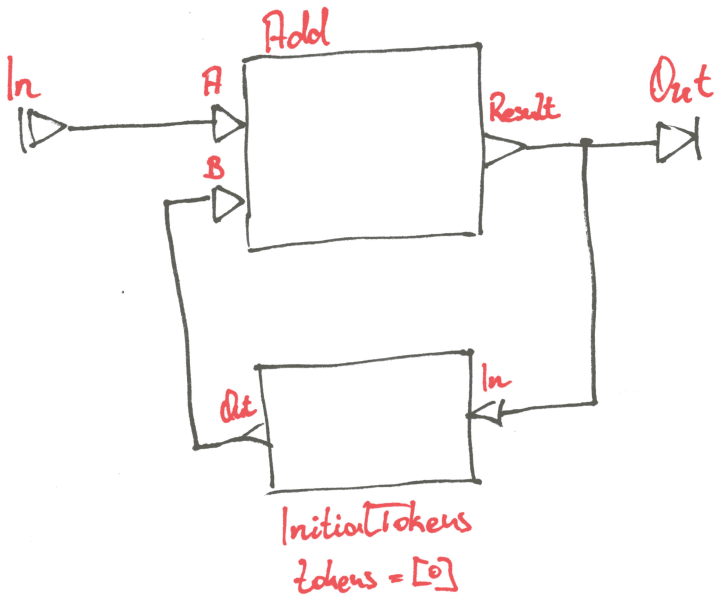

Figure 4.1: Structure of the `Sum` actor network.

From this, we can build the simple network sketched in Fig. 4.1 which generates the sum of all tokens it has seen so far:

```
network Sum () In ==> Out:
entities
  init0 = InitialTokens(tokens = [0]);
  add = Add();
structure
  In --> add.A;
  init0.Out --> add.B;
  add.Result --> init0.In;
  add.Result --> Out;
end
```

[1] This actor uses a construct that has not previously been discussed, involving the `repeat` keyword. Suffice it to say that it is used to read or write a variable number of tokens in one firing. In this case, the number is equal to the length of the argument list `tokens`, and the token values themselves are taken from that list. For more details, cf. the [1].

The definition of a network starts similar to that of an actor, with its name (`Sum`), parameters (none in this case), as well as input and output ports (`In` and `Out`, respectively).

This is followed by a section (introduced by `entities`) instantiating the actors of the network. In this case there are two: one that produces an initial token (the current sum, starting at 0), and another one which is the adder. Note how a parameter is passed to the `InitialTokens` actor by giving the parameter name and then specifying a value to be bound to that parameter.

The next section, introduced by the keyword `structure` builds the network structure, one connection at a time. In this case, it connects the network's `In` port to the `A` port of the `add` actor, then the `Out` port of the `init0` actor to the `B` port of the `add` actor, and finally the `Result` port of the `add` actor to both the `In` input port of the `init0` actor (this is the feedback path), and the `Out` port of the network itself.

Note that actor instances are referred to by their *instance names*, i.e. the names on the left-hand sides of the instantiation expression in the `entities` section—*not* for example by the name of the actor class. So it must be `add.Result` and `init0.In`, as opposed to the **incorrect** `Add.Result` and `InitialTokens.In`.

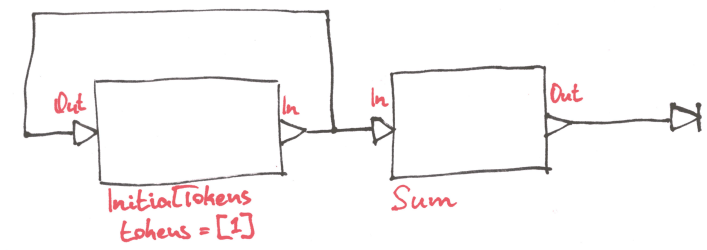

Figure 4.2: Structure of the `Nats` actor network.

Now we can build a network generating the natural numbers, shown in Fig. 4.2:

```
network Nats () ==> Out:
entities
  ones = InitialTokens(tokens = [1]);
  sum = Sum();
structure
  ones.Out --> ones.In;
  ones.Out --> sum.In;
  sum.Out --> Out;
end
```

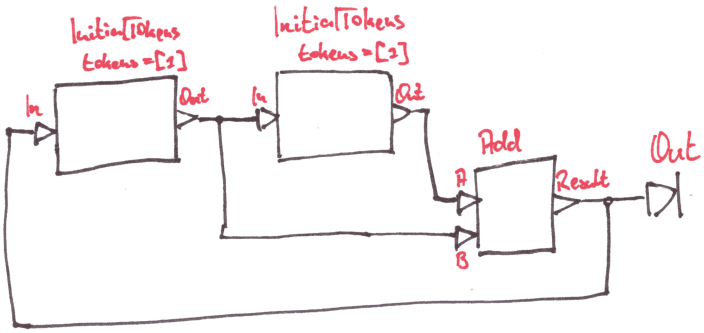

Figure 4.3: Structure of the `Fibs` actor network, generating the Fibonacci numbers.

Similarly, we can easily build the network in Fig 4.3 that generates the Fibonacci numbers:

```
network Fibs () ==> Out:
entities
  init1 = InitialTokens(tokens = [1]);
  init2 = InitialTokens(tokens = [1]);
  add = Add();
structure
  init1.Out --> init2.In;
  init1.Out --> add.A;
  init2.Out --> add.B;
  add.Result --> init1.In;
  add.Result --> Out;
end
```

Since the effect of the `InitialTokens` actor is essentially to place some tokens into a dataflow connection at the beginning of the execution and not much else, it can be useful to represent it a little more concisely than we did in the previous figures by removing the actor itself and instead showing the tokens it produces initially. Fig. 4.4 shows how this would look for the Fibonacci network in Fig. 4.3.

4.2 Parametric network structures

Suppose we would like to build "larger" Fibonacci-like structures—for instance, one that adds the last three rather than two numbers in the sequence, and that starts with three rather than two values of `1`. Such a network would look like the one in Fig. 4.5, and the NL program to create would be this:

```
network Fibs3 () ==> Out:
```

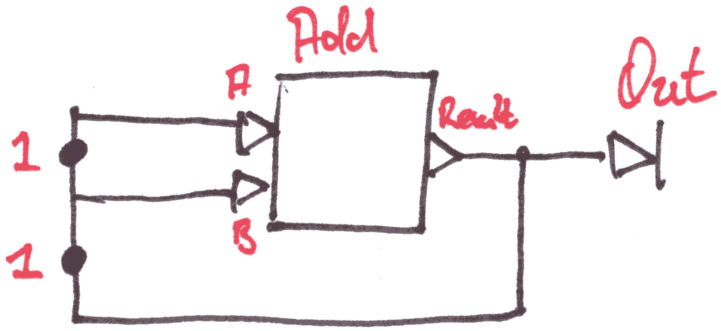

Figure 4.4: An alternative representation of the `Fibs` network in Fig. 4.3, in which instances of the `InitialTokens` actor are represented by the tokens the produce initially.

```
entities
  init1 = InitialTokens(tokens = [1]);
  init2 = InitialTokens(tokens = [1]);
  init3 = InitialTokens(tokens = [1]);
  add1 = Add();
  add2 = Add();
structure
  init1.Out --> init2.In;
  init2.Out --> init3.In;

  init1.Out --> add1.A;
  init2.Out --> add1.B;

  add1.Result --> add2.A;
  init3.Out --> add2.B;

  add2.Result --> init1.In;
  add2.Result --> Out;
end
```

The same construction principle could be extended arbitrarily. Fig. 4.6 shows a 5-stage generalized Fibonacci number generator, which would be generated by the following NL program:

```
network Fibs5 () ==> Out:

entities
  init1 = InitialTokens(tokens = [1]);
  init2 = InitialTokens(tokens = [1]);
```

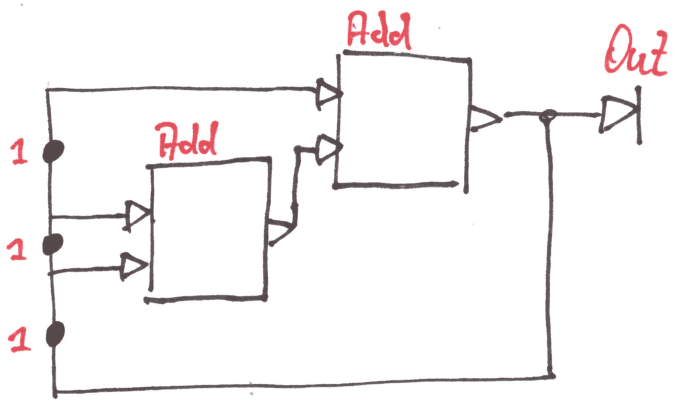

Figure 4.5: Structure of the `Fibs3` network.

```
  init3 = InitialTokens(tokens = [1]);
  init4 = InitialTokens(tokens = [1]);
  init5 = InitialTokens(tokens = [1]);
  add1 = Add();
  add2 = Add();
  add3 = Add();
  add4 = Add();

structure
  init1.Out --> init2.In;
  init2.Out --> init3.In;
  init3.Out --> init4.In;
  init4.Out --> init5.In;

  init1.Out --> add1.A;
  init2.Out --> add1.B;

  add1.Result --> add2.A;
  init3.Out --> add2.B;

  add2.Result --> add3.A;
  init4.Out --> add3.B;

  add3.Result --> add4.A;
  init5.Out --> add4.B;
```

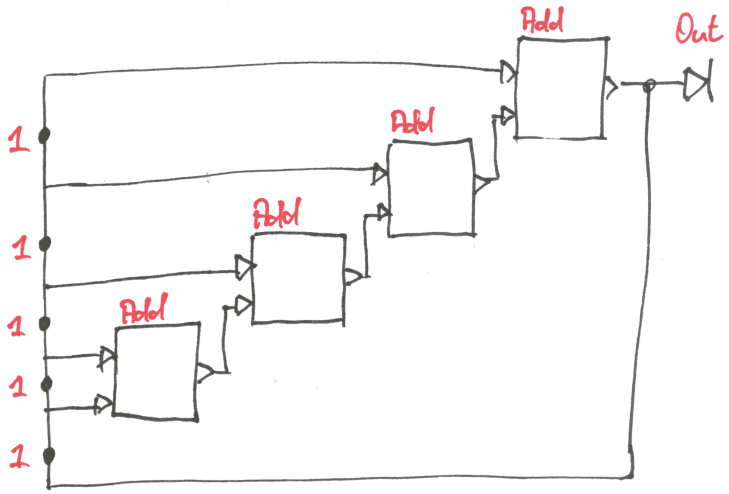

Figure 4.6: Structure of the `Fibs5` network.

```
  add4.Result --> init1.In;
  add4.Result --> Out;
end
```

It would of course be much more convenient to write *one* NL program for the entire family of generalized Fibonacci number generators, which would accept a parameter defining the number of stages. For the usual Fibonacci sequence, that number would be 2, for the two examples above it would be 3 and 5, respectively.

The NL network language provides a number of constructs that allow the construction of parametric network structures. In fact, one way to think about an NL program is as a way of computing a network structure from a set of parameter values. In the previous section, the networks just happened to not have any parameters, and the structures were always the same. However, this is not the case for the following NL program:

```
network FibsN (N) ==> Out:
  entities
    init = [InitialTokens(tokens = [1]) : for i in 1 .. N];
    add = [Add() : for i in 2 .. N];

  structure
    for i in 1 .. N-1 do
      init[i-1].Out --> init[i].In;
    end

    for i in 0 .. N-2 do
```

```
      if i = 0 then
        init[0].Out --> add[i].A;
      else
        add[i-1].Result --> add[i].A;
      end
      init[i+1].Out --> add[i].B;
    end

    add[N-2].Result --> init[0].In;
    add[N-2].Result --> Out;
end
```

This `FibsN` network computes a different network structure for each value of its parameter `N`, and for the values 3 and 5 it computes the structures in Figs. 4.5 and 4.6, respectively.

In the `entities` section, the number of actors instantiated for this network needs to depend on the parameter, `N`, so we see two instantiation expressions that look very much like list comprehensions, one of length `N` (the one for the `InitialTokens` actors), the other of length `N-1` (for the `Adder`s). This is because we need one fewer adder than we have stages for computing a generalized Fibonacci sequence.

Next, in the `structure` section we need to form the appropriate connections. Of course, the number of connections depends on the number of actors to be connected, so we use loop constructs to express the connectivity. The syntax of the loops is not unlike the one in CAL, with the difference that the basic "statement" is not an assignment, but a single connection between two ports.

The first loop hooks up the chain of `InitialTokens` actors, connecting the `Out` port of one actor with the `In` port of the next.

The second loop defines the inputs of each `Add` actor. Usually, these come from the `Result` port of the previous `Add` actor and the `Out` port of one of the `InitialTokens` actors. However, for the *first* adder in the chain, for which there is no previous adder, the input comes from the first `InitialTokens` actor instead. This is what the *conditional structure statement* inside the loop is for.

Finally, the output of the last adder is fed back to the first `InitialTokens` actor, and is also sent to the network's `Out` port.

Appendix A

Running dataflow models

A.1 Prerequisites

The simulation and interpretation tool is based Java 5, avaiable from `http://java.sun.com`.

The model classes (`.cal` and `.nl` files) are located just like Java class files (`.class`) relative to the Java classpath. So when invoking the simulator, the directory tree containing the models needs to be in the classpath.

A.2 Simulation command

The basic class to invoke using the Java interpreter and its parameters are as follows:

`net.sf.caltrop.cli.Simulator` *`[options] model`*

These are the options:

`-D` *`var=expr`* Define the model parameter *var* to be of value *expr*.

`-i` *`file`* Specifies a stimulus file.

`-o` *`file`* Specifies an output file. If the file is ".", the output of the model is printed onto `stdout`.

`-n` *`integer`* Defines an upper bound for the number of steps for which the model is executed. If the model is not dead before reaching that number of steps, the simulation halts.

`-t` *`real`* Defines an uppoer bound for the time stamp associated with any event that is simulated. If the model is not dead before the time stamp of the next event exceeds that number, the simulation halts.

`-ea` Turns on assertion checking. Actor invariants and action pre- and post-conditions are checked, and if violated a warning is produced.

`-bq` *`integer`* Turns on big queue warnings. Whenever any queue exceeds the specified size (i.e. the number of tokens in it is greater than the specified integer), a warning is produced.

Bibliography

[1] Johan Eker and Jörn W. Janneck. CAL language report. Technical Memo UCB/ERL M03/48, Electronics Research Lab, University of California at Berkeley, December 2003. 24, 25, 31

[2] Jörn W. Janneck. NL—a network language. Technical memorandum, ASTG, Processing Solutions Group, Xilinx Inc., 2006. 30