5 Dataflow

Solutions

1. Suppose V is some set and $S = V^{**}$ is the set of finite and infinite sequences of elements of V. This exercise explores some of the properties of the CPO S^n with the pointwise prefix order, for some non-negative integer n. These properties are useful for understanding firing rules.

   (a) Show that any two elements $a, b \in S^n$ that have an upper bound have a least upper bound.

   Solution: Let c be an upper bound of a and b. Then $a \sqsubseteq c$ and $b \sqsubseteq c$. Under the pointwise prefix order, this implies that $\pi_i(a) \sqsubseteq \pi_i(c)$ and $\pi_i(b) \sqsubseteq \pi_i(c)$ for each $i \in \{1, \cdots, n\}$. Since $\pi_i(a)$ and $\pi_i(b)$ are ordinary sequences, if they are both prefixes of the same sequence $\pi_i(c)$, then it must be that either $\pi_i(a) \sqsubseteq \pi_i(b)$ or $\pi_i(b) \sqsubseteq \pi_i(a)$. We can construct a $d \in S^n$ where $\pi_i(d)$ is defined to be $\pi_i(b)$ if $\pi_i(a) \sqsubseteq \pi_i(b)$, and is defined to be $\pi_i(a)$ otherwise, for each $i \in \{1, \cdots, n\}$. Then clearly d is an upper bound of a and b, and moreover, $\pi_i(d) \sqsubseteq \pi_i(c)$ for each $i \in \{1, \cdots, n\}$, so d is a least upper bound under the pointwise prefix order.

   (b) Let $U \subset S^n$ be such that no two distinct elements of U are joinable. Prove that for all $s \in S^n$ there is at most one $u \in U$ such that $u \sqsubseteq s$.

   Solution: Note first that the theorem is trivially true for $n = 0$. For $n > 0$, assume to the contrary that you have two distinct $u, u' \in U$ such that $u \sqsubseteq s$ and $u' \sqsubseteq s$ for some $s \in S^n$. Then s is an upper bound for $\{u, u'\}$. From part (a), $\{u, u'\}$ has a least upper bound, and hence u and u' are joinable, contradicting the assumption that no two distinct elements of U are joinable.

   (c) Given $s \in S^n$, suppose that $Q(s) \subset S^n$ is a joinable set where for all $q \in Q(s)$, $q \sqsubseteq s$. Then show that there is an s' such that $s = (\bigvee Q(s)).s'$.

   Solution: It is sufficient to show that $\bigvee Q(s) \sqsubseteq s$. Note first this is trivially true for $n = 0$, so we henceforth assume $n > 0$. Consider each dimension $i \in \{1, \cdots, n\}$. For each such i, there

is a $q \in Q(s)$ such that $\pi_i(\bigvee Q(s)) = \pi_i(q)$. We know that $\pi_i(q) \sqsubseteq \pi_i(s)$, so we conclude that $\pi_i(\bigvee Q(s)) \sqsubseteq \pi_i(s)$ for each such i. Hence, $\bigvee Q(s) \sqsubseteq s$.

2. Consider the model shown in Figure 5.1. Assume that data types are all $V = \{0,1\}$. Assume f is a dataflow actor that implements an identity function and that Const is an actor that produces an infinite sequence $(0,0,0,\cdots)$. Obviously, the overall output of this model should be this same infinite sequence. The box labeled g indicates a composite actor. Find firing rules and a firing function g for the composite actor to satisfy the generalized firing rules. Note that the composite actor has one input and two outputs.

Solution: Let $U = \{(0),(1),\perp\}$ be the set of firing rules. Note that subsets $\{(0),\perp\}$ and $\{(1),\perp\}$ are joinable. Notice that the greatest lower bound of each of these sets is $\perp$, so the first part of rule 3 is satisfied. Let g be defined so that

$$g((0)) = ((0),\perp) \tag{5.1}$$
$$g((1)) = ((1),\perp) \tag{5.2}$$
$$g(\perp) = (\perp,(0)). \tag{5.3}$$

Note that this firing function yields, as desired, and infinite sequence $(0,0,0,\cdots)$. Note now that if $u = (0)$ and $u' = \perp$, then

$$g(u).g(u') = g(u').g(u).$$

The same is true if $u = (1)$ and $u' = \perp$, so the rest of rule 3 is satisfied.

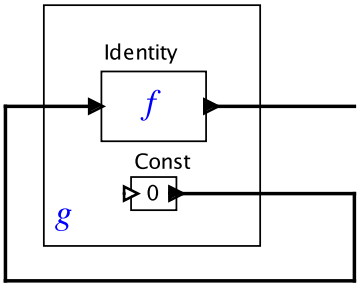


Figure 5.1: A model.

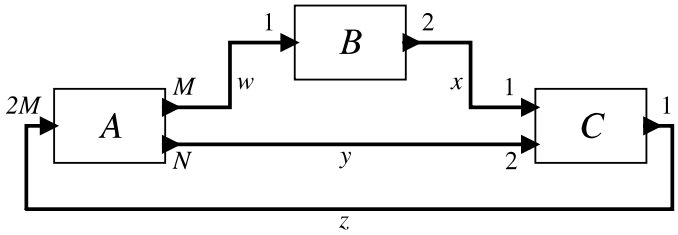


Figure 5.2: A model.

3. Consider the SDF graph shown in Figure 5.2. In this figure, A, B, and C are actors. Adjacent to each port is the number of tokens consumed or produced by a firing of the actor on that port, where N and M are variables with positive integer values. Assume the variables w, x, y, and z represent the number of initial tokens on the connection where these variables appear in the diagram. These variables have non-negative integer values.

   (a) Derive a simple relationship between N and M such that the model is consistent, or show that no positive integer values of N and M yield a consistent model.

Solution: The balance equations are

$$\begin{aligned} Mq_A &= q_B \\ 2q_B &= q_C \\ Nq_A &= 2q_C. \end{aligned}$$

From these we can determine that $N = 4M$ results in a consistent model.

   (b) Assume that $w = x = y = 0$ and that the model is consistent and find the minimum value of z (as a function N and M) such that the model does not deadlock.

Solution: The solution is $z = 2M$. The minimum solution to the balance equations yields $q_A = 1$, regardless of the value of M or N (for a consistent model). The minimum number of initial tokens that enables this is $z = 2M$.

   (c) Assume that $z = 0$ and that the model is consistent. Find values for w, x, and y such that the model does not deadlock and $w + x + y$ is minimized.

Solution: We need to be able to execute C at least $2M$ times to avoid deadlock. Hence $w = 0$, $x = 2M$, and $y = 4M$ will work. However, so will $w = M$, $x = 0$, and $y = 4M$. The latter has a lower value for $w + x + y$. There are no more possibilities, so this latter value is the solution.

   (d) Assume that $w = x = y = 0$ and z is whatever value you found in part (b). Let b_w, b_x, b_y, and b_z be the buffer sizes for connections w, x, y, and z, respectively. What is the minimum for these bus sizes?

Solution: The minimum positive integer solution to the balance equations is $q_A = 1$, $q_B = M$, and $q_C = 2M$. With this solution, the schedule that minimizes the buffer sizes interleaves the executions of B and C. The resulting buffer sizes are $b_w = M$, $b_x = 2$, $b_y = 4M = N$, and $b_z = 2M$.

3 Dataflow

Solutions

1. The multirate actors described in the box on page 105 and the array actors described in the boxes on page 86 and 84 are useful with SDF to construct **collective operations**, which are operations on arrays of data. This exercise explores the implementation of what is called an **all-to-all scatter/gather** using SDF. Specifically, construct a model that generates four arrays with values:

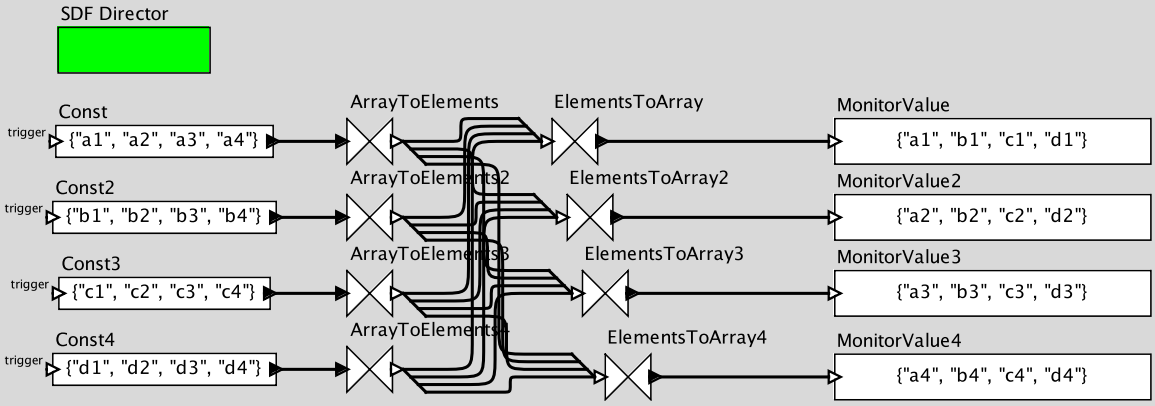
```
{"a1", "a2", "a3", "a4"}
{"b1", "b2", "b3", "b4"}
{"c1", "c2", "c3", "c4"}
{"d1", "d2", "d3", "d4"}
```

and converts them into arrays with values

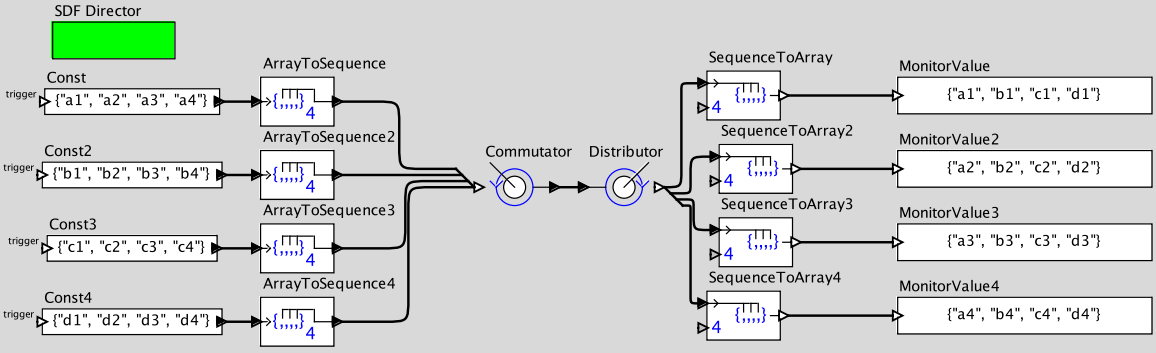
```
{"a1", "b1", "c1", "d1"}
{"a2", "b2", "c2", "d2"}
{"a3", "b3", "c3", "d3"}
{"a4", "b4", "c4", "d4"}
```

Experiment with the use of ArrayToElements and ElementsToArray, as well as ArrayToSequence and SequenceToArray (for the latter, you will also likely need Commutator and Distributor). Comment about the relative merits of your approaches. **Hint:** You may have to explicitly set the channel widths p.63 of the connections to 1. Double click on the wires and set the value. You may also experiment with MultiInstanceComposite.

Solution: Below is an implementation using ArrayToElements and ElementsToArray:



Below is an implementation using ArrayToSequence and SequenceToArray:

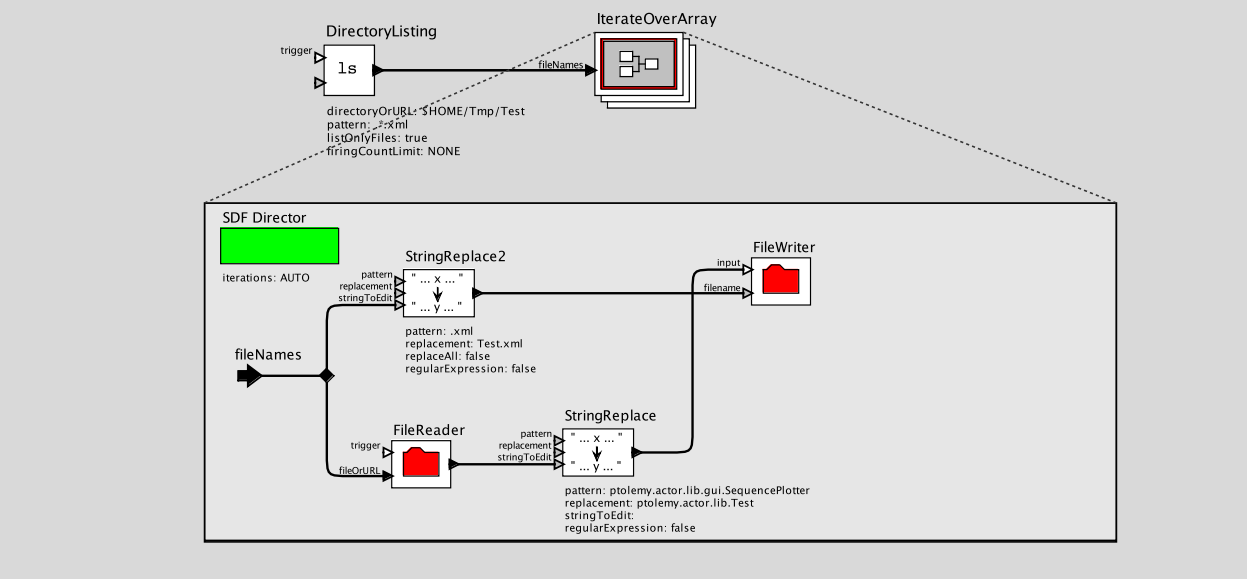


In the above, the Distributor is set to use a *blockSize* of 4.

The latter solution has the advantage of more readable connections, at least in this graphical syntax.

2. Consider the model in Figure 3.15, discussed in Example 3.15. Implement this same model using the IterateOverArray actor and only the SDF director instead of the DDF director (see Section 2.7.2).

Solution: A solution is shown below:



3. The DDF director in Ptolemy II supports an actor called ActorRecursion that is a recursive reference to a composite actor that contains it. For example, the model shown in Figure 3.1 implements the sieve of Eratosthenes, which finds prime numbers, as described by **?**.

   Use this actor to implement a composite actor that computes Fibonacci numbers. That is, a firing of your composite actor should implement the firing function $f : \mathbb{N} \to \mathbb{N}$ defined by, for all $n \in \mathbb{N}$,

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

   When ActorRecursion fires, it clones the composite actor above it in the hierarchy (i.e., its container, or its container's container, etc.) whose name matches the value of its *recursionActor* parameter. The instance of ActorRecursion is populated with ports that match those of that container. This actor should be viewed as a highly experimental realization of a particular kind of higher-order actor. It p.79 is a higher-order actor because it is parameterized by an actor that contains it. Its implementation, however, is very inefficient. The cloning of the actor it references on each firing is expensive in terms of both memory and time. A better implementation would use an approach similar to the stack frame approach used in procedural programming languages. Instead, the approach it uses is more like copying the source code at run time and then interpreting it. In an attempt to make execution more efficient, this actor avoids creating the clone if it has previously created it. Also, the visual representation of the recursive reference is inadequate. There is no way, looking only at the image in Figure 3.1, to tell what composite actor the ActorRecursion instance references. Thus, you cannot really read the program from its visual representation.

   Solution: A solution is shown below, where the ActorRecursion actor references the composite that contains it: