A STRUCTURED DESCRIPTION OF DATAFLOW ACTORS AND ITS APPLICATION

Johan Eker
Jörn W. Janneck

Memorandum No. UCB/ERL M03/13
Electronics Research Laboratory
University of California at Berkeley
13 May 2003

A structured description of dataflow actors and its application

Johan Eker[1] and Jörn W. Janneck[2]

[1] Research, Ericsson Mobile Platforms AB
22183 Lund, Sweden
johan.eker@emp.ericsson.se
[2] EECS Department, University of California at Berkeley
Berkeley, CA 94720, U.S.A.
janneck@eecs.berkeley.edu

Abstract. Many embedded systems have significant parts that are best conceptualized as *dataflow* systems, in which *actors* execute and communicate by sending each other packets of data. This paper proposes a structured view of these actors that focuses on their dataflow characteristics, and it sketches a notation that directly represents this view. It then shows how exploiting this actor structure opens new opportunities for efficiently implementing dataflow systems, demonstrating a technique that distributes actors into several concurrent threads and that synthesizes the required synchronization code. As a result, actor descriptions become more reusable, and reasonably efficient implementations are possible for a larger class of dataflow models.

1 Introduction

Many modern embedded systems contain significant subsystems that are best conceptualized as *dataflow* systems, focusing on the flow of information between their components. There are numerous different dataflow models of computation, most of which fall into one of three categories [11]: Kahn process networks [9], Dennis dataflow [3], and dataflow synchronous languages [1]. This paper focuses on the second kind of dataflow, whose distinguishing characteristic is its notion of *firing*, i.e. atomic steps performed by the components of a model.

Various kinds of dataflow models are used to describe a wide variety of software systems, including signal processing algorithms, control systems, communication systems (such as routers), and even graphical user interfaces and desktop applications.

In all dataflow models of computation, the model components (which we call *actors*, and which might also be called *processes* in other models of computation) communicate by sending each other packets of data called *tokens* along unidirectional channels with exactly one reader and one writer.

A dataflow model such as the one in Fig. 1 represents the potential data dependencies among different parts of a software system (its actors). However, it does not provide information about the behavior of the actors themselves, such as when and how many tokens of data they consume and produce during their execution. As a result, general dataflow models do not provide much information about how the execution of an actor

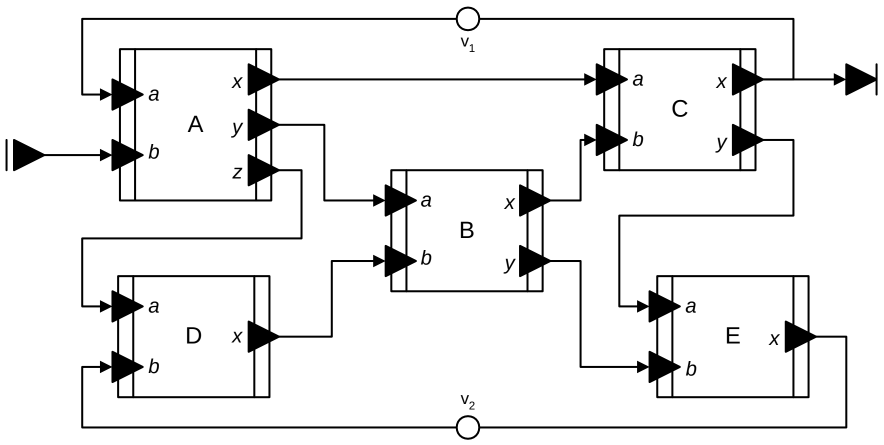

Fig. 1. A dataflow model—actors with ports, the directed connections between their ports, and initial tokens.

depends on the execution of other actors it communicates with—scheduling cannot be statically determined, and consequently, implementations tend to be inefficient.

This has led to the development of a number of specialized dataflow formalisms, or *models of computation*, which are often geared towards a specific application domain—such as synchronous dataflow (SDF) [10], Kahn process networks [9], cyclo-static dataflow (CSDF) [5], and Boolean dataflow (BDF) [2]. These models impose restrictions on what actors are allowed to do—e.g., actors may only consume and produce a constant number of tokens per activation, they must use blocking read operations to access input tokens and the like. The resulting static analysis either allows significantly more efficient implementations (as in SDF, CSDF) and/or provides assurances about the resulting software system (e.g. bounded buffer usage for SDF, determinacy for Kahn process networks).

However, specialized dataflow models of computation have two important drawbacks. First, they usually require actors to be described in a way that is designed to match their specific requirements, making it difficult to use these actors in any other context, such as e.g. another dataflow model of computation. More importantly, they are often very rigid, and cannot be adapted to and used in circumstances that need more flexibility—e.g., synchronous dataflow requires actors to consume and produce a constant number of tokens at each activation, which makes it impossible to directly provide irregular occurring input data to such an actor.

This paper proposes a way of structuring the description of actors so that they can be used more flexibly in dataflow models, while still retaining the efficiency of more constrained dataflow models of computation whenever possible. It presents a small example that showcases some of the structural analyses and subsequent use of the obtained results in generating code from a dataflow model.

Section 2 introduces our actor model, and a structured notation for actors. It also illustrates how these descriptions can be used to manipulate actors and to analyze them. Section 3 applies the notation and the associated techniques to the example in Fig. 1, generating reasonably efficient code for it. We conclude with a discussion of the results and an outlook to further work.

2 Describing dataflow actors

Before we can discuss dataflow models, we need to be more specific about the components of these models, the dataflow actors. In this section we first present a simple formal model for dataflow actors as a foundation for the subsequent discussion. Then we identify an important criterion for structuring actor descriptions, and sketch a notation for dataflow actors that reflects this structure.

2.1 Preliminaries

In the following, we assume a *universe* of all token values $\mathcal{U}$ that can be exchanged between actors. The communication along each connection between actors can be viewed as a sequential stream of tokens, and actors will remove tokens from this stream and add tokens to it. We define the set $\mathbb{S} =_{def} \mathcal{U}^*$ as the set of all finite sequences over $\mathcal{U}$. The set $\mathbb{S}_\infty =_{def} \mathbb{S} \cup \mathbb{S}^{\mathbb{N}}$ is the set of all finite and infinite sequences over $\mathcal{U}$. We write the empty sequence as λ. The length of a finite sequence s is denoted by $\mid s \mid$.

The elements of $\mathbb{S}_\infty$ (and consequently $\mathbb{S}$) are partially ordered by their *prefix relation*: $s \sqsubseteq r$ iff s is a prefix of r, i.e. r starts with the tokens of s in the order they occur in s. For example, `abc` $\sqsubseteq$ `abcd`, and `ab` $\sqsubseteq$ `ab`, but `abc` $\not\sqsubseteq$ `cabd`. Note that for any $s \in \mathbb{S}_\infty$, $\lambda \sqsubseteq s$ and $s \sqsubseteq s$.

Many actors have multiple input and output sequences, so most of the time we will work with tuples of sequences, i.e. with elements of $\mathbb{S}^n$ or $\mathbb{S}^n_\infty$ for some n. The prefix order extends naturally to tuples as follows:

$$(s_i)_{i=1..n} \sqsubseteq (r_i)_{i=1..n} \Leftrightarrow_{def} \forall i = 1..n : s_i \sqsubseteq r_i$$

Note that since $\mathbb{S}_\infty$ and $\mathbb{S}^n_\infty$ under the prefix order both have a least element, and every chain in them has a least upper bound, they are complete partial orders. This property is relevant when relating dataflow models with a notion of firing to Kahn process networks, as in [11, 6].

In the following we use *projections* from $\mathbb{S}^m$ to $\mathbb{S}^n$, with $m \geq n$. These are functions that extract from an m-tuple an n-tuple such that for each value its number of occurrences in the argument is not smaller than the number of occurrences in the result. We write projections using the letter π with appropriate subscripts, and use them to map the input and output of an actor onto a subset of its ports. A special kind of projection maps a tuple onto one port, say p, of the actor. We write this projection as π_p.

2.2 Dataflow actors

We can now define a dataflow actor as an entity that makes atomic steps, and that in each such step consumes and produces a finite (and possibly empty) sequence of tokens at each input or output port. It also has a state (which we will not further characterize), which may change during this step.

Definition 1 ((Dataflow) actor, transition) *Let $\mathcal{U}$ be the* universe *of all values, and $\mathbb{S} = \mathcal{U}^*$ be the set of all finite sequences in $\mathcal{U}$. For any non-empty set Σ of* states *an* m-to-n dataflow actor *(or just* actor *for short, when m and n are understood or not relevant) is a labeled transition system*

$$\langle \sigma_0, \Sigma, \tau, \succ \rangle$$

with $\sigma_0 \in \Sigma$ its initial state, *and*

$$\tau \subseteq \Sigma \times \mathbb{S}^m \times \mathbb{S}^n \times \Sigma$$

its transition relation. *An element of τ is called a* transition. *Finally, $\succ$ is a non-reflexive, anti-symmetric and transitive partial order relation on τ, called its* priority *relation.*

For any transition $(\sigma, s, s', \sigma') \in \tau$ we also write

$$\sigma \xrightarrow[\tau]{s \mapsto s'} \sigma'$$

or, if τ, or s and s' are understood or not relevant,

$$\sigma \xrightarrow{s \mapsto s'} \sigma' \quad \text{or} \quad \sigma \xrightarrow[\tau]{} \sigma' \quad \text{or} \quad \sigma \longrightarrow \sigma'$$

calling σ (σ') a direct predecessor (successor) *of σ' (σ), and s (s') the* input (output) *of the transition. Together, (s, s') are the transition* label.

The set of all m-to-n actors with firing is $\mathcal{A}^{m \longrightarrow n}$. The set of all actors is

$$\mathcal{A} =_{def} \bigcup_{m,n \in \mathbb{N}} \mathcal{A}^{m \longrightarrow n}$$

The core of this definition can be found in [11], where “firing rules” defined the input tuples and a firing function mapped those to output tuples. State was added in an extension proposed in [6]. Here, we add the priority relation, which makes actors much more powerful, by e.g. allowing them to test for the absence of tokens. On the other hand, it can make them harder to analyze, and it may introduce unwanted non-determinism into a dataflow model. We will see further uses of this facility in the context of the example.

Intuitively, the priority relation determines that a transition cannot occur if some other transition is possible. We can see this in the definition of a valid *step* of an actor, which is a transition such that two conditions are satisfied: the required input tokens must be present, and there must not be another transition that has priority.

Definition 2 (Enabled transition, step) *Given an m-to-n dataflow actor* $\langle \sigma_0, \Sigma, \tau, \prec \rangle$*, a state* $\sigma \in \Sigma$ *and an* input tuple $v \in \mathbb{S}^m$*, a transition* $\sigma \xrightarrow[\tau]{s \mapsto s'} \sigma'$ *is* enabled *iff*

$$s \sqsubseteq v$$

$$\neg \exists \sigma \xrightarrow{r \mapsto r'} \sigma'' \in \tau : r \sqsubseteq v \wedge \sigma \xrightarrow{s \mapsto s'} \sigma' \succ \sigma \xrightarrow{r \mapsto r'} \sigma''$$

A step *from state* σ *with input* v *is any enabled transition* $\sigma \xrightarrow{s \mapsto s'} \sigma'$*. The* residual input tuple v' *is defined by* $v = s + v'$*.*

Note that the second condition for a transition to be enabled becomes vacuously true if $\succ = \emptyset$, leaving $s \sqsubseteq v$, the usual dataflow condition. We call an actor with an empty priority relation a *pure* dataflow actor.

As an example, consider a possible definition for actor `C` from the example in Fig. 1, as depicted in Fig. 2a. After defining the actor name and its input and output ports, its description consists of two procedures: the first, `canFire`, determines whether the conditions are fulfilled to fire this actor. In the example, this is the case if there is 1 input token available on each input port. The procedure `step` specifies what happens when the actor fires. For this actor, this depends on the value of one of the input tokens. In either case, one token is produced on the `x` output port, but on the `y` output port we may or may not see a token produced. This actor has no state, or rather it has precisely one, its initial state σ_0.

(a)

```
actor $C$
  $a, b \Longrightarrow x, y$ :
  $canFire$ :
    return $hasToken(a, 1)$
           and $hasToken(b, 1)$
  $step$ :
    $v := get(a)$
    $w := get(b)$
    if $q(v)$
      $put(x, C_1(v, w))$
    else
      $put(x, C_2(v, w))$
      $put(y, v)$
```

(b)

```
actor $C$
  $a, b \Longrightarrow x, y$ :
  $one$ : action $a : [v], b : [w] \Longrightarrow$
                 $x : [C_1(v, w)]$
        guard $q(v)$
  $two$ : action $a : [v], b : [w] \Longrightarrow$
                 $x : [C_2(v, w)], y : [v]$

  priority $one > two$
```

Fig. 2. The procedural (a) and action-structured (b) description of actor `C` in Fig. 1.

2.3 An action structure for actors

While the description in Fig. 2a is relatively straightforward, it does not match our definition of a dataflow actor very well—the token production and consumption of a transition are implicit, and the result of a potentially very complex program. This is unfortunate, because this information is crucial for many applications, and should thus be easy to extract from an actor description, both for human readers and for automatic tools.

We therefore propose a notation that is closer to the structure of Def. 1, and that represents externally observable actor behavior (i.e. its token consumption and production) more directly.[1] The representation of `C` in this notation is shown in Fig. 2b. The body of this description consists of two *actions*, each of which specifies part of the transition relation. Each action consists of (a) *input patterns*, describing how many tokens are to be read from each input port, and what these are called in the rest of the action, (b) *output expressions* computing the output values as a function of the input (and the state, if the actor has state, which `C` does not), and (c) a *guard*, which is a Boolean expression that must be true for the action to be enabled. Actions also can contain a *body*, which defines modifications of the actor state. They may optionally be labeled (by ”one” or ”two” in the example).

For instance, the following actor produces the running sum of its input:

```
actor $Sum$
  $v \Longrightarrow sum$ :
  $s := 0$
  action $v : [a] \Longrightarrow sum : [s]$
  do
     $s := s + v$
```

This action defines a family of transitions of the form

$$[s \mapsto n] \xrightarrow{v \mapsto n+v} [s \mapsto n + v]$$

where $[s \mapsto n]$ is the state of the actor binding n to the actor variable s.

An action represents a set of transitions, and the union of these sets is the transition relation of the actor. As a result, the lexical ordering of actions inside an actor description is irrelevant. Therefore, non-determinism is easily expressed by several actions:

```
actor $Merge$
  $a, b \Longrightarrow out$ :
  action $a : [v] \Longrightarrow out : [v]$
  action $b : [v] \Longrightarrow out : [v]$
```

If we want to express that the transitions defined by an action have priority over those defined by another action, we need to do so explicitly, using the *priority* construct, as shown in Fig. 2b[2]—it specifies that the transitions resulting from the action labeled ”one” have priority over those resulting from action ”two,” thereby piecewise

[1] The notation proposed here is a simplified form of the CAL actor language, defined in [4].

[2] In this case, an equivalent result could have been obtained by guarding the second action by $\neg q(v)$. Our choice reflects good specification style, and was also intended to showcase a common use of priorities.

constructing the priority relation of the actor.[3] A biased merge actor could look like this:

```
actor $BiasedMerge$
  $a, b \Longrightarrow out:$
  $A$ : action $a : [v] \Longrightarrow out : [v]$
  $B$ : action $b : [v] \Longrightarrow out : [v]$
  priority $A > B$
```

Actors can of course consume (or produce) any number of tokens in one firing. The following actor, for example, consumes two and distributes them to its two output ports:

```
actor $DistributeTwo$
  $a \Longrightarrow out1, out2 :$
  action $a : [v1, v2] \Longrightarrow out1 : [v1], out2 : [v2]$
```

Not every conceivable actor can be described by a finite number of these actions, and neither by the richer actions supported in the CAL actor language. However, those that cannot be represented turn out to be rather unusual cases.[4]

Before we turn back to our example, the next section discusses a few simple concepts and operations on actors, and how they relate to action-structured descriptions.

2.4 Properties, projections, and partitions

For many applications, we need to rely on actors having specific properties—for instance, if we want to apply synchronous dataflow techniques to a dataflow model (static scheduling and buffer allocation), the actors in the model need have constant token production and consumption rates. This property can be formulated as follows:

$$\begin{aligned} \forall p \in P_{in} : \exists k \in \mathbb{N} : \forall \sigma \xrightarrow{s \mapsto s'} \sigma' \in \tau : \ \mid \pi_p(s) \mid = k \\ \wedge \forall p \in P_{out} : \exists k \in \mathbb{N} : \forall \sigma \xrightarrow{s \mapsto s'} \sigma' \in \tau : \ \mid \pi_p(s') \mid = k \end{aligned}$$

Where P_{in} and P_{out} are the sets of input and output ports, respectively.[5] This property is obvious from an action-structured description of an actor: we only need to count the input and output tokens at each port in each action.

In many cases, such a property may not hold for the entire actor, but only for parts of it. For example, actor `C` is not a synchronous dataflow actor, because sometimes it produces an output token at `y`, and sometimes it does not. But assume we use `C` in a synchronous dataflow model without actually connecting its `y` output to anything,

[3] There is a subtle technicality due to the fact that in principle a transition can result from more than one action. In this case, the highest priority is chosen—cf. [4] for details.

[4] See [8] for details on the different kinds of actions and actors.

[5] In Def. 1, ports are identified by position, hence we may think of the port sets as sets of natural numbers. Actor descriptions, of course, use symbolic names for ports, but we simply assume that there exists some mapping from these names to the respective position in the input or output tuple, and take the liberty to use the symbolic constants whenever this is convenient.

effectively discarding whatever it produces—in that case, we could use C inside an SDF model, because it does have constant token rates on all its other ports.

The notion we employed here is that of an *actor projection*, i.e. the view of an actor through a subset of its ports. We can formally describe it as follows:

Definition 3 (Actor projection) *Given an actor* $\langle \sigma_0, \Sigma, \tau, \succ \rangle$*, and two projections* (π_{in}, π_{out}) *on its input and output ports, we define an* actor projection

$$\langle \sigma_0, \Sigma, \tau', \succ' \rangle =_{def} \langle \sigma_0, \Sigma, \tau, \succ \rangle \downarrow_{(\pi_{in}, \pi_{out})}$$

as follows. For any transition in τ*, we define its projection*

$$(\sigma, s, s', \sigma') \downarrow_{(\pi_{in}, \pi_{out})} =_{def} (\sigma, \pi_{in}(s), \pi_{out}(s'), \sigma')$$

and then we define the projected transition relation and the priority relation like this:

$$\tau' = \left\{ t \downarrow_{(\pi_{in}, \pi_{out})} \mid t \in \tau \right\}$$
$$t_1 \downarrow_{(\pi_{in}, \pi_{out})} \succ' t_2 \downarrow_{(\pi_{in}, \pi_{out})} \Leftrightarrow_{def} t_1 \succ t_2 \wedge t_1 \downarrow_{(\pi_{in}, \pi_{out})} \neq t_2 \downarrow_{(\pi_{in}, \pi_{out})}$$

Note that it is necessary to eliminate those cases where the projection of two different transformations yields the same result, because otherwise $\succ'$ would no longer be irreflexive.

We can easily see how to perform projection on an action-structured description—we just eliminate the corresponding parts in the input patterns and output expressions, along with the respective port declaration. For instance, projecting actor C in Fig. 2b onto its input ports a and b and its output port x results in the following actor description:

```
actor $C - abx$
  $a, b \Longrightarrow x$ :
  $one$ : action $a : [v], b : [w] \Longrightarrow$
                 $x : [C_1(v, w)]$
         guard $q(v)$
  $two$ : action $a : [v], b : [w] \Longrightarrow$
                 $x : [C_2(v, w)]$

  priority $one > two$
```

This is clearly an SDF actor.[6] A projection that results in an SDF actor is *maximal* if there is no larger set of ports such that projecting onto it would also result in an SDF actor.

Not all such projections yield legal actor descriptions. For example, the complementary projection onto just the y port results in the following actor:

[6] In practice, depending on the language constructs it offers, an actor language may allow the static recognition of larger classes of actors, e.g. cyclo-static dataflow actors, which also support static schedules, or other efficient implementation techniques. We use SDF as an example because it is simple, well-understood, widely used, and can be recognized without introducing more complex language features.

```
**actor** $C - y$
    $\Longrightarrow y :$
    $one$ : **action** $\Longrightarrow$
            **guard** $q(v)$        $//\, undefined\, variable$
    $two$ : **action** $\Longrightarrow y : [v]$        $//\, undefined\, variable$

    **priority** $one > two$
```

This actor is illegal, because it contains references to an undefined variable (its definition in an input pattern had been removed).

Occasionally, actors exhibit a kind of internal concurrency in the sense that actions only interact via the actor state, but not by using the same ports. Consider the following description of actor B from Fig. 1:

```
**actor** $B$
    $a, b \Longrightarrow x, y :$
    $s := init_B$
    **action** $a : [v] \Longrightarrow x : [B_1(v, s)]$
    **do**
        $s := B_2(v, s)$

    **action** $b : [v] \Longrightarrow y : [B_3(v, s)]$
    **do**
        $s := B_4(v, s)$
```

Its two actions use disjoint sets of ports, even though they both affect and depend on the actor state. (This is of course the reason why these two actions need to be realized in the same actor, rather than just in two independent actors: actors cannot share state.) When implementing this actor, we could run the two actions in different threads on different processors, as long as we make sure that they do not both access the state at the same time. Therefore, determining sets of independent ports can be crucial to efficiently implementing a network of dataflow actors.

We call these sets an *actor partitioning*, which we define as follows.

Definition 4 (Port dependency, actor partitioning) *Given an actor with input ports P_{in}, output ports P_{out}, and transition relation τ, we define a* dependency *relation $\cong$ between ports in $P_{in} \cup P_{out}$ as the reflexive, and transitive closure over the following* direct dependency *relation $\overset{1}{\cong}$:*

$$p_i \overset{1}{\cong} q_i \Leftrightarrow_{def} \exists(\sigma, s, s', \sigma') \in \tau : \pi_{p_i}(s) \neq \lambda \wedge \pi_{q_i}(s) \neq \lambda$$
$$p_o \overset{1}{\cong} q_o \Leftrightarrow_{def} \exists(\sigma, s, s', \sigma') \in \tau : \pi_{p_o}(s') \neq \lambda \wedge \pi_{q_o}(s') \neq \lambda$$
$$p_i \overset{1}{\cong} p_o \Leftrightarrow_{def} \exists(\sigma, s, s', \sigma') \in \tau : \pi_{p_i}(s) \neq \lambda \wedge \pi_{p_o}(s') \neq \lambda$$
$$p_o \overset{1}{\cong} p_i \Leftrightarrow_{def} p_i \overset{1}{\cong} p_o$$

for all $p_i, q_i \in P_{in}$ and $p_o, q_o \in P_{out}$. Clearly, $\cong$ is an equivalence relation over the actor ports.

The set $(P_{in} \cup P_{out})/\cong$ of equivalence classes is called the actor partitioning.

For example, actor C has only one such equivalence class, its actor partitioning is $\{\{a, b, x, y\}\}$. By contrast, actor D has the partitioning $\{\{a, x\}, \{b, y\}\}$. Constructing an actor partitioning from the action-structured description is straightforward: the token rates of all transitions that are represented by an action are identical, therefore we can substitute actions for transitions in the definition above, and easily construct the equivalence relation, and its equivalence classes.

The next section discusses the use of these constructions in the context of deriving an implementation for a dataflow model.

3 Example: Implementing a dataflow model

We now turn to the task of implementing the dataflow model in Fig. 1, assuming the actor definitions in Fig. 3. Of these, actor A is particularly interesting: similar to B, it has a nontrivial partitioning into $\{\{a, b, x, y\}, \{z\}\}$, and both partitions also happen to have constant token rates. But unlike B, the $\{z\}$-partition is a *source* in the sense that it has no input port. Its firing is controlled by the state, which is modified by both actions. The dual of this situation can be found in actor D, whose $\{a\}$-partition is only an input port, and the associated action just modifies the state.

There are many ways of implementing a dataflow model. If it can be statically scheduled, then it is often a good idea to do so, and to turn the model into a sequential program with preallocated buffers.[7] However, for arbitrary dataflow networks, there are typically two extremes: (1) A fully parallel implementation, where each actor is assigned a thread or process, and communication happens exclusively via asynchronous messages that are queued by the receiver until they can be consumed. (2) An entirely sequential simulator, that iterates over the actors in the model and fires them when they are firable, queuing the resulting tokens until they are used.

None of these options are very attractive for most practical cases. The first results in a lot of context switching (or alternatively requires many parallel resources), most of which will be useless because an actor will find that it still does not have enough input tokens to fire. The second variant eliminates any real concurrency from the implementation, which is usually at odds with the choice of a dataflow model for expressing parallel computation in the first place.

However, the results of section 2 allow a somewhat more flexible approach to dataflow implementation, which rests on recognizing partial synchronous dataflow ”islands” that can be scheduled statically, and regions that can be executed concurrently, even when they cut through actors. In these cases, we can derive the appropriate synchronization code to ensure that the atomicity of an actor transition is preserved.

[7] Of course, if such a schedule contains complex computation that is potentially concurrent, it may be preferable to parallelize this part of the schedule. Since this work focuses on the interaction between statically schedulable regions inside an arbitrary dataflow model, we will make the simplifying assumption that we prefer a straightforward sequential implementation for statically scheduled parts of the model.

```
actor $A$
  $a, b \Longrightarrow x, y, z$ :
  $s := init_A$
  action $\Longrightarrow z : [A_1(s)]$
  guard $p(s)$
  do
     $s := A_2(s)$

  action $a : [v], b : [w] \Longrightarrow$
        $x : [A_3(v, w)], y : [A_4(w)]$
  do
     $s := A_5(v, w, s)$
```

```
actor $C$
  $a, b \Longrightarrow x, y$ :
  $one$ : action $a : [v], b : [w] \Longrightarrow x : [C_1(v, w)]$
        guard $q(v)$
  $two$ : action $a : [v], b : [w] \Longrightarrow$
                 $x : [C_2(v, w)], y : [v]$
  priority $one > two$
```

```
actor $E$
  $a, b \Longrightarrow x$ :
  $one$ : action $a : [v], b : [w] \Longrightarrow$
                 $x : [E_1(v, w)]$
  $two$ : action $a : [v] \Longrightarrow x : [E_2(v)]$
  priority $one > two$
```

```
actor $B$
  $a, b \Longrightarrow x, y$ :
  $s := init_B$
  action $a : [v] \Longrightarrow x : [B_1(v, s)]$
  do
     $s := B_2(v, s)$

  action $b : [v] \Longrightarrow y : [B_3(v, s)]$
  do
     $s := B_4(v, s)$
```

```
actor $D$
  $a, b \Longrightarrow x$ :
  $s := init_D$
  action $a : [v] \Longrightarrow$
  do
     $s := v$

  action $b : [v] \Longrightarrow x : [D(v, s)]$
```

Fig. 3. Actor definitions for the model in Fig. 1.

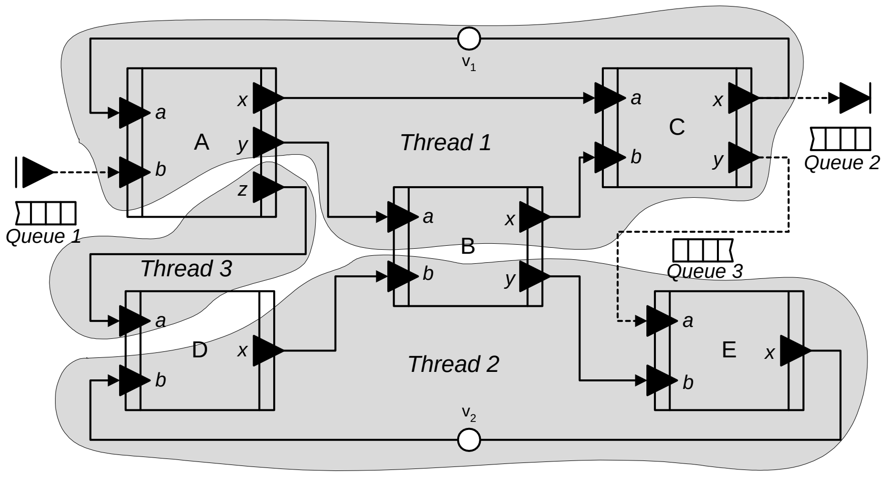

Fig. 4. Partioning of the model in Fig. 1. Dashed lines are non-SDF connections.

Statically schedulable islands. We first determine for each actor maximal subactors that are SDF. In the example, these are $\{A.a, A.b, A.x, A.y\}$ (we use the notation $X.p$ to identify the port p of actor X), $\{A.z\}$, $\{B.a, B.x\}$, $\{B.b, B.y\}$, $\{C.a, C.b, C.x\}$, $\{D.a\}$, $\{D.b, D.x\}$, and $\{E.b, E.x\}$. Considering only connections between these SDF partitions, we compute static schedules for the resulting SDF islands in the model. In the example, we would disregard the connection between $C.y$ and $E.a$ for this purpose, and consider $A.b$ a free input port.

Model partitioning. We partition the model based on actor partitionings. Note that these cannot be smaller than the maximal SDF subactors. Each SDF subactor will be contained by precisely one partition. Conversely, an actor partition may contain any number of SDF subactors, including none. Now we define a *model partition* as follows. Two ports are in the same model partition if

1. They are both in the same actor partition.
2. They are both part of the same SDF island.

The resulting model partitioning is shown in Fig. 4. Each partition can be assigned to a different thread. Communication along connections that are part of an SDF island happens via preallocated buffers (in the example, these are all of length 1, and thus can be represented by a single variable), while all other communication is performed using FIFO queues. Whenever the model partitioning cuts through an actor, it needs to synchronize on a semaphore—in the example, this needs to be done for actors A, B, and D. For simplicity, we assume that queue accesses are automatically synchronized.

Based on this partitioning, and the static schedules within each partition, we can now generate code for each of the threads, which is shown in Fig. 5. The code starts with a few state variables that were part of the actors, and declaring semaphores required for synchronization. We have named variables representing static buffers according to the receiver port—b_E is the buffer in front of port $E.b$. State variables are called s, with the name of the actor as subscript. In the interest of clarity, the code has not been optimized in any way, so that e.g. locks are in fact held longer than strictly necessary.

Note how thread 1 waits for input on the external input queue, and how the code blocks corresponding to the statically scheduled actors follow each other. They are surrounded by lock/unlock pairs if they come from an actor who were cut by the model partitioning. Also worth noting is the way the non-SDF connection is realized by a queue (Queue3) in both thread 1 and 2.

4 Discussion and conclusion

In this paper we propose a new way of structuring and describing dataflow actors that have a notion firing. Our goal is to make the actor descriptions more flexible, and more amenable to analysis. The new notation is designed to directly represent the semantic model we use for dataflow actors. We present a few simple analysis techniques and apply them to the transformation of a dataflow model into reasonably efficient code.

In some ways, the work reported here can be viewed as complementing the work in [7], which describes a semantic framework for composing a network of actors under an arbitrary model of computation into a composite actor. By contrast, this paper focuses on a very specific model of computation, viz. dataflow, and the result of the composition is not another actor, it is a concurrent imperative program.

```
$a_A := v_1$
$b_D := v_2$
$s_A := init_A$
$s_B := init_B$
$s_D := init_D$

semaphore $M_A, M_B, M_D$
```

```
$Thread1$ :
  forever
    wait until $available(Queue1, 1)$
    lock $M_A$
    $w := get(Queue1)$
    $a_C := A_3(a_A, s_A)$
    $a_B := A_4(w, s_A)$
    $s_A := A_5(a_A, w, s_A)$
    unlock $M_A$

    lock $M_B$
    $b_C := B_1(a_B, s_B)$
    $s_B := B_2(a_B, s_B)$
    unlock $M_B$

    if $q(a_C)$
      $val := C_1(a_C, b_C)$
      $a_A := val$
      $put(Queue2, val)$
    else
      $val := C_2(a_C, b_C)$
      $a_A := val$
      $put(Queue2, val)$
      $put(Queue3, a_C)$

```

```
$Thread2$ :
  forever
    lock $M_D$
    $b_B := D(b_D, s_D)$
    unlock $M_D$

    lock $M_B$
    $b_E := B_3(b_B, s_B)$
    $s_B := B_4(b_B, s_B)$
    unlock $M_B$

    if $available(Queue3, 1)$
      $v := get(Queue3)$
      $b_D := E_1(v, b_E)$
    else
      $b_D := E_2(b_E)$

```

```
$Thread3$ :
  forever
    lock $M_A$
    $doFire := p(s_A)$
    if $doFire$
      $val := A_1(s_A)$
      $s_A := A_2(s_A)$
    unlock $M_A$

    if $doFire$
      lock $M_D$
      $s_D := val$
      unlock $M_D$
```

Fig. 5. The generated code corresponding to the partitioning in Fig. 4: global variable initializations and code for each of the three threads.

The proposed actor notation makes actor descriptions more flexible mainly for two reasons. (1) It is easy to identify subactors that have specialized desirable properties, such as constant token rates. (2) It is straightforward to exploit internal parallelism, making it easy to slice actor descriptions and distribute them over several threads, while synthesizing the synchronization code. To our knowledge, this has not been done before for dataflow programs.

Because of this flexibility, we can produce reasonable code for dataflow models that are not quite statically schedulable, but that have significant parts that are, and that communicate with each other, or with the outside world asynchronously through FIFO queues. Many practical applications fall into this category, such as e.g. signal processing algorithm that are asynchronously configured by outside events (e.g. caused by user input, power events etc.), or network processors that occasionally classify, drop, or put back packets based on header information, content, or time. The framework presented here allows them to express these features without compromising the efficiency of the generated code, or the analyzability of the overall model.

Future work will address more detailed analysis of actors and models, provide more control over how models are partitioned, and produce more efficient code. For instance, thread 3 in the example keeps polling the state s_A, which can be very inefficient, particularly because it needs to lock the semaphore M_A during this time. A more efficient implementation would put this thread to sleep until the relevant state is modified, and then wake it up. Easily recognizing optimization potential such as this makes an action-structured actor notation useful.

References

1. A. Benveniste, P. Caspi, P. LeGuernic, and N. Halbwachs. Data-flow synchronous languages. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency—Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1994.
2. Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*. PhD thesis, University of California at Berkeley, 1993. also Technical Report UCB/ERL 93/69.
3. J. B. Dennis. First version data flow procedure language. Technical Memo MAC TM 61, MIT Lab. Comp. Sci., May 1975.
4. Johan Eker and Jörn W. Janneck. CAL actor language—language report. Technical Report UCB/ERL 03/TBD, University of California at Berkeley, 2003. preliminary version on http://www.gigascale.org/caltrop.
5. Marc Engels, Greet Bilsen, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow: Model and implementation. In 1994, editor, *28th Annual Asilomar Conference on Signals, Systems, and Computers*, pages 503–507, October-November.
6. Jörn W. Janneck. *Syntax and Semantics of Graphs—An approach to the specification of visual notations for discrete event systems*. PhD thesis, ETH Zurich, Computer Engineering and Networks Laboratory, July 2000.
7. Jörn W. Janneck. Actors and their composition. Technical Report UCB/ERL 02/37, University of California at Berkeley, 2002.
8. Jörn W. Janneck. A taxonomy of dataflow actors. Technical Report UCB/ERL 03/TBD, University of California at Berkeley, 2003.
9. Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress*. North-Holland Publishing Co., 1974.
10. E. Lee and D. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, pages 55–64, September 1987.
11. Edward A. Lee. A denotational semantics for dataflow with firing. Technical Report UCB/ERL M97/3, EECS, University of California at Berkeley, January 1997.