Static Sessional Dataflow

Dominic Duggan and Jianhua Yao

Department of Computer Science
Stevens Institute of Technology
Hoboken, New Jersey 07030, USA
{dduggan,jyao1}@stevens.edu

Abstract. Sessional dataflow provides a compositional semantics for dataflow computations that can be scheduled at compile-time. The interesting issues arise in enforcing static flow requirements in the composition of actors, ensuring that input and output rates of actors on related channels match, and that cycles in the composition of actors do not introduce deadlock. The former is ensured by flowstates, a form of behavior type that constrains the firing behavior of dataflow actors. The latter is ensured by causalities, a form of constraints that record dependencies in the firing behavior. This article considers an example variant of the sessional dataflow approach for dataflow applications, expressing known ideas from signal processing in a compositional fashion.

1 Introduction

Dataflow has an honored tradition in declarative parallel programming [12,10]. It has renewed significance today, given the importance attached to deterministic parallelism as a way of coping with the challenges of scalable parallel programming. Many of the applications of parallel processing are in stream processing, e.g., streaming multimedia data, again motivating interest in dataflow processing. Part of the challenge of dataflow processing is in scheduling the execution of dataflow graphs without unbounded buffering of data between actors in the net. In signal processing, synchronous dataflow has enjoyed some success for multi-rate applications, with many variations of the basic idea developed over the years [13].

The purpose of sessional dataflow is to provide a compositional semantics for dataflow computations that can be scheduled at compile-time. To explain why compositionality is important, in synchronous dataflow and its variants, a dataflow graph is described in terms of atomic actors, and flow edges connecting them. A compositional semantics allows both atomic actors, and subnets resulting from the composition of actors, to be viewed uniformly as dataflow actors. Compositionality is obviously important for scaling dataflow programming. The interesting issues arise in enforcing static flow requirements in the composition of actors, ensuring that input and output rates of actors on related channels match, and that cycles in the composition of actors do not introduce deadlock. Ultimately the purpose of sessional dataflow is to support dynamic operations on subnets, including update and reconfiguration, while ensuring that assumptions underlying static scheduling are not violated by these operations.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 484–508, 2012.
© Springer-Verlag Berlin Heidelberg 2012

In the embedded systems and digital signal processing community, a very useful class of restricted Kahn networks has been identified, the so-called *synchronous dataflow (SDF)* [13] networks. SDF networks assumed fixed static input and output rates for actors when they fire in a dataflow network. Such networks are important because they can be scheduled statically by the compiler, ensuring a fixed upper bound on the amount of buffer space needed. Numerous extensions of SDF have been defined over the years, all pushing the envelope of expressivity while remaining in the space of dataflow applications that can be scheduled statically.

More recently, new domain-specific languages such as Streamit [16] have been defined, based on the principles of SDF, but also providing support for compiling programmer code to run on modern parallel architectures. Streamit is a dataflow language intended for the efficient compilation of stream processing programs. Its design rationale is that of *structured dataflow*. Rather than allowing arbitrary dataflow graphs, Streamit imposes structure on the graph, in order to facilitate compiler analysis and optimization. This is enforced by only supporting certain forms of nodes in a dataflow graph.

In this work, we consider another form of dataflow language. We go back to Kahn's original idea, of a dataflow network consisting of a collection of software components that communicate asynchronously via buffered message-passing. Deterministic parallelism is provided by preventing contention for message channels, and by preventing components from polling message channels. As with Kahn's original proposal, our core language is a conventional imperative language. We impose a type system on this language that ensures the static behavior required for compile-time scheduling, as with SDF. This allows the incremental construction of dataflow graphs as composite actors, based on connecting input and output channels in two graphs (that may be the same graph). There are two components to the compositional description: a notion of *flowstate*, analogous to typestate in object-oriented languages, that captures the static message-passing behavior of a process, and a notion of *causalities*, that allows the liveness of a dataflow graph to be checked compositionally even while channels are encapsulating in composite graphs.

In Sect. 2 we introduce sessional dataflow with the interface and implementation specification for a simple (atomic) actor. This relates the constraints on actor behavior reflected in an actor interface, with the actual internal implementation of the actor that is encapsulated by this interface. These two are not traditionally related in work on synchronous dataflow and its derivative techniques, where actors are treated as "black boxes" and the actor code left unanalyzed. In Sect. 3 we consider a compositional approach to building actor implementations, based on binding communication channels between two existing actors. As an exercise in statically ensuring that the composition of actors is well-formed, we require that the result of composing actors into a dataflow net, effectively a composite actor, be statically schedulable. We provide a type system in Sect. 4 and an operational semantics in Sect. 5. Sect. 6 considers related work, while Sect. 7 provides our conclusions.

2 Actors

In this section, we provide an example of a form of static dataflow that has been found useful for parallel processing in signal processing and embedded systems. So-called synchronous dataflow (SDF), more aptly named *static dataflow*, assumes that on each actor "firing," a statically fixed number of inputs is consumed on each input channel and a statically fixed number of outputs is produced on each output channel. With this restriction on a fixed number inputs and outputs for each firing, and a further requirement that there be no cyclic data dependencies in the graph connecting the actros, the scheduling of a synchronous dataflow net can be performed by the compiler. All scheduling decisions, and the amount of buffer space required, are determined at compile-time.

An actor specification needs a few other aspects to be defined. Although firing is atomic in SDF, our semantics for firing is implemented in a C-like core language, that consumes and produces messages one at a time. For modeling the states of an actor, we use the notion of *flowstate*, that tracks the state of an actor during a firing cycle. In addition, we need a specification of the input and output channels of an actor, that will subsequently be coupled with channels for other actors to form a dataflow network. An example of a specification for an actor in our type system is provided by the following:

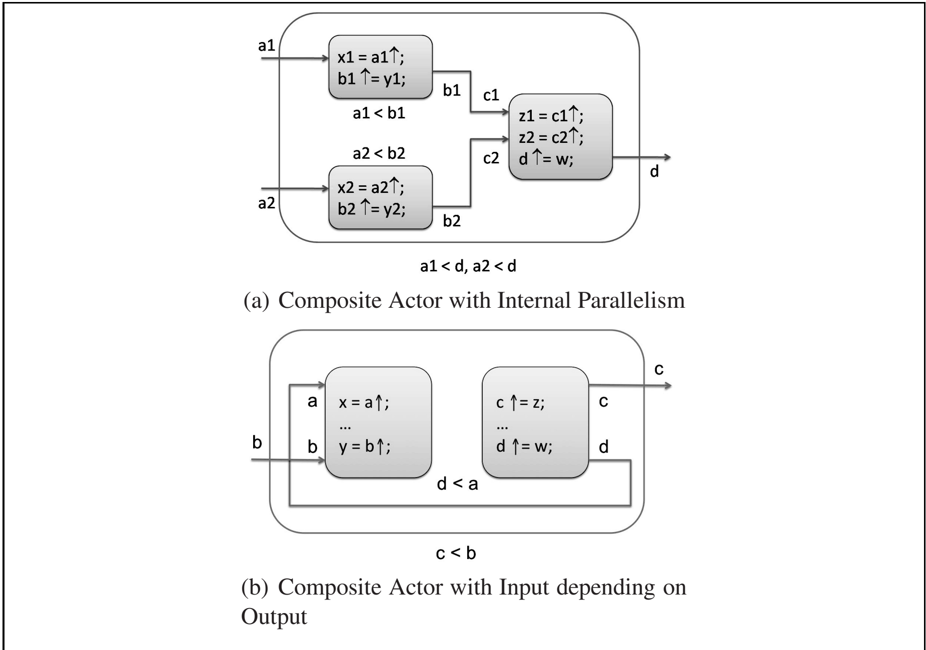
```
actor interface IActor
{
   in channel<float> a;
   in channel<float> b;
   out channel<float> c;
   causality a < c, b < c;
   flowstate 3'a, b, 2'c.
}
```

This is the expression of an actor type in our system. The type specifies input and output communication channels, and allowable communications on those channels using a flowstate specification. The flowstate rule in the example above requires that the actor consume three inputs on the `a` channel and one input on the `b` channel, and produces two outputs on the `c` channel. In what order should these inputs and outputs be performed? It is tempting to restrict firings so that all inputs are consumed before any outputs are produced, but once we compose actors into composite actors (dataflow nets), it is no longer possible to ensure this. Even if we restricted actors to only inputs or only outputs, but not a mixture of the two, we could still have scenarios where the consumption of an input in one actor depended on the production of an output in another actor. Therefore we must allow for aribtirary interleavings of inputs and outputs, while avoiding deadlock where for example an output channel and input channel are linked to the same underlying channel.

Therefore we enrich actor interfaces with a notion of *causalities*. This is demonstrated in the example above, where the causalities specify that outputs on channel `c` depend causally on inputs on channel `a` and on channel `b` ($a < c$ and $b < c$). The exact number of inputs is provided by the multiplicities in the flowstate.

Why provide the causalities, since in this example all of the outputs depend on all of the inputs? This will be true in general for simple atomic actors, but may not be

true once we compose actors into nets of arbitrary complexity, with subnets running in parallel. The causalities are then useful to ensure that connecting an input and an output channel in such a composite actor does not introduce deadlock in the execution of the dataflow graph.



(a) Composite Actor with Internal Parallelism

(b) Composite Actor with Input depending on Output

Fig. 1. Sequential and Parallel Inputs

Fig. 1(a) and Fig. 1(b) demonstrate two composite actors. Fig. 1(a) provides a composite actor where two actors on the left consume inputs in parallel, and these are then consumed in a particular order by the actor on the right. We define causalities to reflect the fact that message sending is asynchronous, so in some sense the outputs of an actor, once their causally preceding input events occur, may occur in an indeterminate order. Fig. 1(b) provides another composite actor, one where the output on channel `c` must causally precede the input on channel `a`, since the output on internal channel `d` causally precedes the input on internal channel `a`. Outputs are parallel despite the fact that they are produced by a single sequential thread.

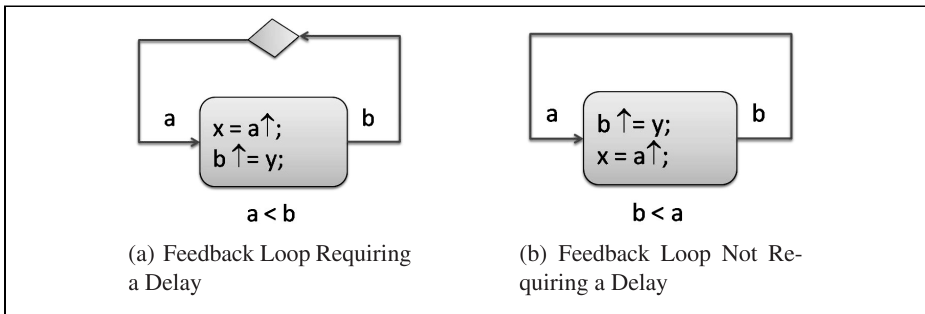
Our actor semantics is effectively a limited form of *cyclostatic dataflow* [2]. In the latter, an actor has a finite state control logic, and transitions between states of this logic on each firing. Its firing pattern then depends on the current state that it is in. Because we are providing specifications for input consumption and output production at the level of individual communication steps, the semantics of a “firing” in the traditional SDF sense is non-atomic, and we are essentially tracking a finite state control logic in the process of a firing. We consider how the language can be extended to cyclostatic firing at the end of Sect. 4.

The specifications of the input-output behavior make no reference to the actual values that are transmitted. For simplicity we have assumed that the channel types are fixed, so that only values of the declared type may be transmitted on a channel. In practice it may be beneficial to relax this restriction, though we defer these considerations to future work. We comment further on this and other future extensions in Sect. 6.

An implementation of this actor specification uses a conventional programming language to define the actor behavior, in the style of Kahn's original proposal for dataflow networks:

```
actor Actor implements IActor
{
   float x1, x2, x3, y1;
   loop {
      x1 = a↓; x2 = a↓; x3 = a↓; y1 = b↓;
      c ↑ (x1+x2); c ↑ (y1+x3);
   }
}
```

The definition of the actor implementation inherits the interface specification: input and output channels, causalities and flowstates. The operation for reading from an input channel c is denoted by $c\downarrow$, while the operation of writing a value (asynchronously) to an output channel is denoted by c $\uparrow$ v. The body of the actor is otherwise conventional C code, except for the top-level loop construct that guarantees the actor is always able to offer the specified firing behavior. The flowstate in the actor specification establishes behavior obligations for its execution, subject to the constraints imposed by the causalities.



(a) Feedback Loop Requiring a Delay

(b) Feedback Loop Not Requiring a Delay

Fig. 2. Causality and Feedback

Fig. 2 clarifies the point of the causalities. In general the issue is to detect when connecting two open channels in the same actor may introduce a cycle in the dependencies between the channels. To avoid this cycle which woud lead to deadlock, the connection of the channels is required to introduce a “delay,” by filling the buffer for the channel with default initial values. Fig. 2(a) illustrates this, where the single output channel of an actor is connected to its input channel. The actor first reads from the input channel

a, before outputing to the output channel b. Note that we do not try to track data flow dependencies, our interest is in the control flow dependency from the consumption of input on a to the production of output on b. Suppose these two open channels are connected to the same shared channel, We assume an obvious causality from the output end of a shared channel to the input end, so this binding will introduce the causality $b < a$. This will introduce a cycle in the causalities, which we cannot allow. Therefore in this case the connection of two channels a and b on the same underlying channel must include a delay, as indicated by the diamond in Fig. 2(a).

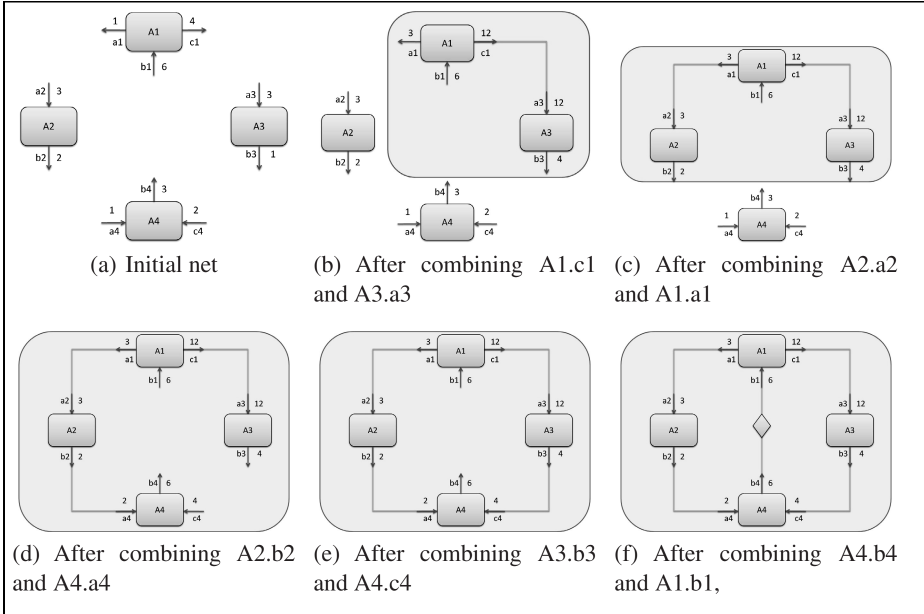
In the example in Fig. 2(b), on the other hand, the appending of data to the output buffer is done before input is performed. This results in the causality $b < a$ for the actor body. This does not necessarily mean that data flows from the output event to the input event, but there is at least a causal dependency, in that the occurrence of the output event is a prerequisite for the occurrence of the input event. When these input and output open channels are connected using the same shared input channel, then the output produced on the output channel does indeed propagate to the input channel to be consumed, but this is immaterial as far as scheduling is concerned, since communication is strictly internal to the actor. The causality $b < a$ that is added as a result of this binding of channels b and a adds no further constraints, since there is already a dependency from b to a, and no scheduling cycle is introduced, so a delay is not necessary.

3 Dataflow Nets

In the previous section, we considered the "programming-in-the-small" aspects of ensuring that an actor satisfied its behavior specification. In this section, we consider the "programming-in-the-large" aspect of ensuring that the composition of actors is in some sense well-formed. We consider the case of ensuring that the composition of synchronous dataflow actors is schedulable, based on the static firing rates of the actors.

In general, the approach to composition of actors is to provide a binary connection operation for linking the output data channel on one actor with the input data channel of another. We denote this operation by `connect(A.a,B.b)`. Here it is important to distinguish between open channels and shared channels. An *open channel* is one of the form described in the previous section, a channel that is declared in an actor interface, and referred to in an actor body by operations for consuming messages and appending messages to message buffers.

For deterministic semantics, it is important that there be no nondeterministic contention for access to a channel. For example, a nondeterministic merge might be provided by allowing multiple actors to send simultaneously to the same merge channel. Synchronization on access to the channel, performed by the compiler and runtime system, could ensure that the message append operations are atomic. However the order in which messages are appended would be nondeterministic, based on dynamic scheduling of actors and interleaving of their multithreaded executions. While nondeterministic merge is a useful operation in some cases, our intention is to establish a baseline that ensures deterministic execution, before considering later how to extend this with nondeterminism.



(a) Initial net
(b) After combining A1.c1 and A3.a3
(c) After combining A2.a2 and A1.a1
(d) After combining A2.b2 and A4.a4
(e) After combining A3.b3 and A4.c4
(f) After combining A4.b4 and A1.b1,

Fig. 3. Dataflow nets

Our approach is to ensure exclusive access to a communication channel between two actors, the one actor sending on that channel and the other actor receiving on that channel. The connection operation `connect(A.a,B.b)` creates a new private communication channel, binds the `a` output channel on the `A` actor to the output part of this new private channel, and binds the `b` input channel on the `B` actor to the input part of this new private channel. We refer to such a private channel as a *shared channel*. Since (for now) we provide no way for an actor to send any of its communication channels to another actor, exclusive access by a pair of actors to a shared channel is ensured.[1]

What is the result of connecting actors? The semantics should be compositional, so that the connection of two actors should be indistinguishable to outside observers from a single actor. For synchronous dataflow, the only part of the outside interface of note for a combined actor is the remaining open channels after a connection, and the firing rates for those channels.

A type system for interconnection should guarantee that the actors being combined are in some sense compatible, so that the resulting actor is statically schedulable. The firing rates for actors that are interconnected may be different on the channel on which they are connected. The purpose of static scheduling is to match the relative input and output rates of communicating actors during execution. For connection of distinct

[1] Actor interfaces include polarity information about access to channels, and the connection operation requires that the accesses by the actors be complementary. It is indeed possible that the actors being connected are the same. The connection operation requires knowledge of when it is the case that the same actors are being connected, as we will see.

actors, it is always the case that their firing rates are compatible: Simply adjust their relative firing rates so that, on the connecting channel, their rates are the least common multiple of the original rates on that channel.

Fig. 3(a) depicts four characters: `A1`, `A2`, `A3` and `A4`. Each of these actors has open channels with firing rates. For example, actor `A1` has open channel `a1` with firing rate 1, open channel `b1` with firing rate 2, and open channel `c1` with firing rate 4. The names of these channels outside the actor are not significant, and we assume for simplicity that all channels are renamed apart.

In Fig. 3(b), the actors `A1` and `A3` are connected by binding the open channels `A1.c1` and `A3.a3` along an anonymous shared channel. Since the output rate of `A1` does not match the input firing rate of `A3`, we adjust the firing rates of the two actors to make them compatible. The connection of an output channel of `A1` to the input channel of `A3` causes the addition of the causality `c1` < `a3`. Although `a3` is elided in the resulting interface (since it has been bound to the output end of a communication channel), transitive closure of causalities adds the constraint `b1` < `b3` (from `b1` < `a3` and `a3` < `b3`).

Fig. 3(c) depicts the result of connecting the composite actor `connect(A1.c1,A3.a3)` with the actor `A2`, by binding the open channels `connect(A1.c1,A3.a3).a1` and `A2.a2` to a shared channel. In this case, the data rates of the actors on the respective open channels match, so no adjustment of firing rates is necessary.

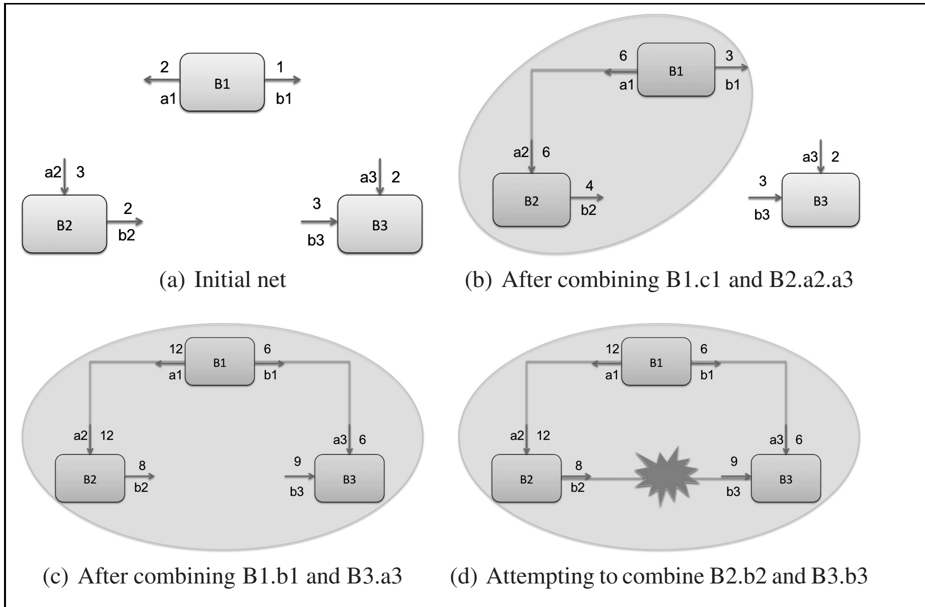
Fig. 3(d) depicts the result of connecting the composite actor from Fig. 3(c) with the actor `A4`. This connection is done on the `b2` and `a4` open channels. Because of the difference in data rates, the firing rates for `A4` must be adjusted

At this point, we have only one (composite) actor, with some remaining open channels. We now complete the net by connecting different channels within the same actor. Fig. 3(e) depicts the result of connecting the `b3` and `c4` channels on this composite actor.

This actor can be completed by forming a feedback loop by connecting the output of the `b4` channel to the input of the `b1` channel. The types, and in particular the inclusion `b4` < `24`, reveals the existing data dependency from the output channel `b4` to the input channel `b1`. This data dependency, revealed in the inclusion constraint in the interface, signals that linking the output channel to the input channel will in this case introduce a feedback loop. In order to ensure that the resulting net does not deadlock, a *delay* must be introduced in this new channel connection, as depicted by the diamond in Fig. 3(f).

What could possibly go wrong? Fig. 4 demonstrates how the checking during the composition of actors may fail. We have three actors, `B1`, `B2` and `B3`, with data rates as described in Fig. 4(a). We compose `B1` and `B2` by binding the channels `B1.a1` and `B2.a2` to a shared channel, adjusting the firing rates as necessary to make their data rates on the shared channel match. We repeat this exercise by composing the resultant composite actor with `B3`, binding the `b1` and `a3` channels to a shared channel. At this point, there is a problem: It is not possible to bind the remaining open channels `b2` and `b3` to each other, because they have different data rates.

To analyse the problem, we note that in general the scheduling of SDF actors can be thought of as the solution of a homogeneous system of linear equations. The independent variables in this system of equations are the number of times each actor fires on an iteration of the dataflow net, and the equations specify the constraint that the amount of



(a) Initial net (b) After combining B1.c1 and B2.a2.a3

(c) After combining B1.b1 and B3.a3 (d) Attempting to combine B2.b2 and B3.b3

Fig. 4. Unschedulable dataflow net

outputs produced by each actor must match the number of inputs consumed, on each net iteration. If this constraint is not satisfied, then some message buffers will grow without bound during the execution of the net.

For the example above, we obtain the following system of equations, where $F_{\mathtt{B}i}$ denotes the number of firings of actor $\mathtt{B}i$ on each iteration of the net:

$$
\begin{aligned}
2 \cdot F_{\mathtt{B2}} - 3 \cdot F_{\mathtt{B1}} &= 0 \\
2 \cdot F_{\mathtt{B3}} - 3 \cdot F_{\mathtt{B2}} &= 0 \\
F_{\mathtt{B3}} - 2 \cdot F_{\mathtt{B1}} &= 0
\end{aligned}
$$

Solving for the independent variables by eliminating $F_{\mathtt{B3}}$, we obtain the equation:

$$3 \cdot F_{\mathtt{B2}} - 4 \cdot F_{\mathtt{B1}} = 0$$

Then using this and the first of the original equations to eliminate $F_{\mathtt{B2}}$, we obtain the equation:

$$9 \cdot F_{\mathtt{B1}} - 8 \cdot F_{\mathtt{B1}} = 0$$

The only solution to this equation is to fire `B1`, and therefore the other actors, zero times, i.e., to never run the dataflow net. This demonstrates the importance of being able to distinguish the cases when we are combining two distinct actors, and when we are connecting two open channels in the same actor. In the former case, when the actors are distinct, we can adjust the actors' firing rates so that the data rates on the connecting

channel are the least common multiple of the data rates on the corresponding open channels in the actors. In the latter case, when we are connecting channels on the same actor, the data rates must match on the corresponding open channels. If they do not, we do not have the extra degree of freedom that we have with distinct actors to adjust firing rates. Indeed, discovering different data rates on the channels is an important part of ensuring, at composition time, that we do not compose two actors into a composite actor (i.e., a dataflow net) that cannot be scheduled.

Variable aliasing is a potentially troublesome issue, for two reasons. First, communication on a channel changes the type of that channel, since the channel type reflects the communications that may be performed on that channel. we avoid the problem of variable aliasing by not allowing aliased references to communication channels. This is compatible with approaches such as for example session types that similarly constrain the bindings of variables to resources whose usage is tracked by linear or affine types.

A second potentially troublesome issue is with the connection of actors. As we have seen, connecting different actors provides a degree of freedom in adjusting the firing rates of the actors so that they match on the channels on which they are connrected. If we allow aliasing of actor references, then we must face the issue of how to deal with scenarios such as the following:

```
IActor2 f (IActor A, IActor B)
     return connect(A.a,B.b);
}
```

How can we prevent a scenario such as `f(A0,A0)`, for some actor `A0` that implements the `IActor` specification? This is a known issue in type systems for safe resource management, as discussed in Sect. 6. In our semantics, we avoid this issue because the `connect` operation makes copies of the two actors being composed. This is a potentially expensive operation, and a better choice of operations might split this into a connection operation that performed update in place on the argument actor specifications, and an explicit `clone` operation for explicitly making a copy of an actor. This choice however makes the actor connection operation a "strong update," modifying the interface of the original actor, which in turn requires ensuring that there be no references to the original actor remaining in the program (including aliases). Our copying semantics avoids this complication.

4 Type System

In this section we consider a core language to support the examples in the previous sections, including a type system to ensure valid program executions. We consider an operational semantics and type soundness in the next section. We name this kernel language $\mathbf{S_{SDF}}$. We only consider synchronous dataflow in this account, but we comment on the extension to the cyclostatic case at the end of this section.

The syntax of types is provided in Fig. 5. For simplicity we assume a single base type of `float`, for floating point values. Similarly we assume that only floating point values are exchanged between actors in each message exchange, so the channel type does not need to describe the type of data exchanged on the channel. Although the polyadic pi-calculus generalizes messages to include tuples of values, this is not necessary in our

$$
\begin{array}{rcl}
T \in \text{Type} & ::= & \mathsf{float} \mid AS \mid \mathsf{channel}\ \pi \\
AS \in \text{Actor sig} & ::= & \mathsf{actsig}(O, FS) \\
K \in \text{Causalities} & ::= & \{\} \mid \{a < b\} \mid K_1 \cup K_2 \\
O \in \text{Open channels} & ::= & \{\} \mid O_1 \cup O_2 \mid \{((\mathbf{c}, c) : \mathsf{channel}\ \pi)\} \\
\pi \in \text{Polarity} & ::= & + \mid - \mid \pm \\
ES \in \text{Event set} & ::= & \{\} \mid \{n \cdot a\} \mid ES_1 \uplus ES_2 \\
FS \in \text{Flowstate} & ::= & ES \mid \{FS \mid K\} \mid (FS_1; FS_2) \mid (FS_1 \parallel FS_2) \mid FS^* \mid FS^\omega
\end{array}
$$

Fig. 5. Abstract syntax of $\mathbf{S_{SDF}}$ Types

current framework because channels are private to a single sender and receiver. We do record polarity information for a channel, which records whether it can be used by that actor for input (polarity +), or for output (polarity −), or both (polarity $\pm$).

The type of interest is that of actors. An *actor signature* has three parts, as we have seen:

1. A *causality set* K is a set of causality constraints between channels, of the form $a < b$, that reflects firing constraints between channels: If $a < b$, then in a firing of the actor or dataflow net, a communication on b depends on a communication on a. For simplicity, we assume that *all* communications on b depend on all communications on a. This set of dependencies must never contain a cycle.
2. A set of *open channels* O. Each element of this set is a triple $((\mathbf{c}, c) : T)$, recording for an "open" channel its channel type. This channel type has the form $\mathsf{channel}\ \pi$, where T is the type of data transmitted on the channel (we only allows floats to be transmitted in this article), and π is the polarity of the open channel. The channel has two names: its *internal name* c by which it is identified internally in the actor, and its *external name* $\mathbf{c}$ by which it is referenced when composing with other actors. We distinguish these names in order to allow renaming apart of internal channel names when actors are composed, without affecting the external interface. The internal and external names in different open channel bindings should obviously be distinct from each other. We define the domain of an open channel set as:
$$dom(O) = \{c \mid ((\mathbf{c}, c) : T) \in O\}.$$
We define the *external domain* of an open channel set as:
$$edom(O) = \{\mathbf{c} \mid ((\mathbf{c}, c) : T) \in O\}.$$
3. The *flowstate* of an actor records its expected firing behavior. The primitive form of an actor flowstate is an *event state*, a multiset of communication events $\{m_1 \cdot a_1, \ldots, m_k \cdot a_k\}$, where each a_i represents a communication event, either a sending of a message on a channel or a receipt of a message on a channel. In either case, we use the channel name to denote the event; whether it is an input or an output event can be determined by the channel's polarity. The multiplicities $m_1, \ldots, m_k$ record the number of occurrences of each event in an actor execution. The remaining forms of flowstate are used to describe the flowstate resulting from joining computations, either sequentially $(FS_1; FS_2)$ or in parallel $(FS_1 \parallel FS_2)$. Note that in the latter case

there may be communication dependencies between the actors running in parallel. The other forms of flowstates are for computations that can repeat an arbitrary number of times (FS^*) and that loop infinitely often (FS^ω). The latter corresponds to the top-level flowstate of an actor, primitive or composite.

$v \in$ Values	$::=$	$n \mid a \mid x$	
$s \in$ Statement	$::=$	$(\mathsf{var}\ x = e;\ s)$	Bind variables
	$\mid$	$\mathsf{if}\ (v)\ s_1;\ \mathsf{else}\ s_2$	Conditional
	$\mid$	$\mathsf{while}\ (v)\ s$	Loop
	$\mid$	$\mathsf{loop}\ s$	Infinite loop
	$\mid$	$\mathsf{fire}_K\ s$	Firing
	$\mid$	skip	Do nothing
	$\mid$	$(s_1; s_2)$	Sequential
$e \in$ Expression	$::=$	$f(v_1, \ldots, v_k)$	Builtin
	$\mid$	$v_1 = v_2$	Assignment
	$\mid$	$\mathsf{run}\ v$	Run a network
	$\mid$	$c \downarrow$	Receive a message
	$\mid$	$c \uparrow v$	Send a message
	$\mid$	$\mathsf{actor}(O, s)$	Atomic actor
	$\mid$	$\mathsf{connect}_{m,n}(v_1.\mathbf{c}_1, v_2.\mathbf{c}_2)$	Connect two actors
	$\mid$	$\mathsf{connectSelf}_m(v_1.\mathbf{c}_1, v_1.\mathbf{c}_2)$	Connect within an actor
	$\mid$	$\mathsf{connectSelfDelay}_m(v_1.\mathbf{c}_1, v_1.\mathbf{c}_2)$	Connect with delay

Fig. 6. Abstract syntax of $\mathbf{S_{SDF}}$ statements

Fig. 6 provides the abstract syntax for programs in $\mathbf{S_{SDF}}$. Values are numbers n, names a (for actors and channels, both allocated on the heap), and variables x. The syntax of statements includes a conditional[2], while loops, and a construct for doing nothing. We also have a construct (fire) for explicitly specifying the firing behavior of an actor body. The construct of interest is the binding statement, which introduces a new variable bound to the result of evaluating a definition. Some of the definitional expressions return a dummy value (the number 0). In these cases, the variable binding expression is used solely to sequence the computation. Why do we also include the sequencing of statements? Our expressional language is in A-normal form, but we have constructs such as the while loop and the conditional that do not fit the definition of an execution step in A-normal form. If we convert this language, both expressions and statements, into A-normal form, we obtain an exponential blow-up in code size unless we residualize the continuation of a conditional. We avoid these complications by including sequencing of statements.

The simplest form of definition is the invocation of builtin functions, presumably to perform arithmetic operations on numeric values. We also allow assignment, though only for values of base type, i.e., floating point values, because of the aliasing issue described in Sect. 3. As expected, we also have operations for receiving and sending messages.

[2] We are assuming that our language has no round-off error. Obviously a more complete language definition would include integers and Booleans.

$$\frac{}{\vdash_\Sigma \{\}\ \mathsf{ok}}\ \textsc{Env Empty} \qquad \frac{\vdash_\Sigma \Gamma\ \mathsf{ok} \quad \Gamma \vdash_\Sigma T}{\vdash_\Sigma \Gamma, x : T\ \mathsf{ok}}\ \textsc{Env Extend}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathsf{ok}}{\Gamma \vdash_\Sigma n : \mathsf{float}}\ \textsc{Const} \qquad \frac{\vdash_\Sigma \Gamma\ \mathsf{ok} \quad (x:T) \in \Gamma}{\Gamma \vdash_\Sigma x : T}\ \textsc{Var} \qquad \frac{\vdash_\Sigma \Gamma\ \mathsf{ok} \quad (a:T) \in \Gamma}{\Gamma \vdash_\Sigma a : T}\ \textsc{Name}$$

$$\frac{\Gamma, K \vdash_\Sigma e : T : FS_1 \quad x \notin dom(\Gamma) \quad (\Gamma \cup \{(x:T)\}), K \vdash_\Sigma s : FS_2}{\Gamma, K \vdash_\Sigma (\mathsf{var}\ x = e; s) : (FS_1; FS_2)}\ \textsc{Bind}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathsf{ok}}{\Gamma, K \vdash_\Sigma \mathsf{skip} : \{\}}\ \textsc{Skip} \qquad \frac{\Gamma \vdash_\Sigma v : \mathsf{float} \quad \Gamma, K \vdash_\Sigma s_1 : FS \quad \Gamma, K \vdash_\Sigma s_2 : FS}{\Gamma, K \vdash_\Sigma (\mathsf{if}\ (v)\ s_1;\ \mathsf{else}\ s_2) : FS}\ \textsc{If}$$

$$\frac{\Gamma \vdash_\Sigma v : \mathsf{float} \quad \Gamma, K \vdash_\Sigma s : FS}{\Gamma, K \vdash_\Sigma (\mathsf{while}\ (v)\ s) : FS^*}\ \textsc{While} \qquad \frac{\Gamma, K \vdash_\Sigma s : FS}{\Gamma, K \vdash_\Sigma (\mathsf{loop}\ s) : FS^\omega}\ \textsc{Loop}$$

$$\frac{\Gamma, K \vdash_\Sigma s : FS}{\Gamma, \{\} \vdash_\Sigma (\mathsf{fire}_K\ s) : \{FS \mid K\}}\ \textsc{Fire}$$

$$\frac{(f : \overrightarrow{\mathsf{float}} \rightarrow \mathsf{float}) \in \Sigma \quad \overrightarrow{\Gamma \vdash_\Sigma v_k : \mathsf{float}}}{\Gamma, K \vdash_\Sigma f(v_1, \ldots, v_k) : \mathsf{float} : \{\}}\ \textsc{Builtin}$$

$$\frac{v_1 \in \{x, l\} \quad \Gamma \vdash_\Sigma v_1 : \mathsf{float} \quad \Gamma \vdash_\Sigma v_2 : \mathsf{float}}{\Gamma, K \vdash_\Sigma v_1 = v_2 : \mathsf{float} : \{\}}\ \textsc{Assign}$$

$$\frac{(c : \mathsf{channel}\ \pi) \in \Gamma \quad \pi \leq - \quad \Gamma \vdash_\Sigma v : \mathsf{float}}{\Gamma, K \vdash_\Sigma c \uparrow v : \mathsf{float} : \{1 \cdot c\}}\ \textsc{Send} \qquad \frac{(c : \mathsf{channel}\ \pi) \in \Gamma \quad \pi \leq +}{\Gamma, K \vdash_\Sigma c \downarrow : \mathsf{float} : \{1 \cdot c\}}\ \textsc{Receive}$$

$$\frac{\Gamma \vdash_\Sigma v : \mathsf{actsig}(\{\}, FS)}{\Gamma, K \vdash_\Sigma \mathsf{run}\ v : \{\}}\ \textsc{Run} \qquad \frac{\Gamma, K \vdash_\Sigma s_1 : FS_1 \quad \Gamma, K \vdash_\Sigma s_2 : FS_2}{\Gamma, K \vdash_\Sigma (s_1; s_2) : (FS_1; FS_2)}\ \textsc{Seq}$$

$$\frac{\Gamma, K \vdash_\Sigma s : FS_0 \quad \Gamma \vdash_\Sigma FS \quad \Gamma, K \vdash_\Sigma FS_0 \cong FS}{\Gamma, K \vdash_\Sigma s : FS}\ \textsc{Stmt Eq}$$

Fig. 7. Type system

The next three definitions are for defining actors: the definition of an atomic actor, and operations for connecting actors on complementary open channels, with and without a delay. An atomic actor has a causality set, open channel set and flowstate specification, as with actor signatures. In addition, the actor has a (single-threaded) actor body, an expression that is constrained by the flowstate specification. The final definition returns no values, but starts the asynchronous execution of an actor that has no remaining open channels.

To describe a type system for this simple minilanguage, we add a type environment Γ, described as follows:

$$\Gamma ::= \{\} \quad | \quad \Gamma_1 \cup \Gamma_2 \quad | \quad \{(a:T)\} \quad | \quad \{(x:T)\}$$

$$\frac{\Gamma_0 = \{(c:T) \mid ((\mathbf{c},c):T) \in O\} \quad \Gamma_0, K_0 \vdash_\Sigma s : FS}{\Gamma, K \vdash_\Sigma \mathsf{actor}(O,s) : \mathsf{actsig}(O,FS) : \{\}} \ \textsc{Actor}$$

$$\frac{\begin{array}{c} \Gamma \vdash_\Sigma v_i : \mathsf{actsig}(O_i, \{ES_i \mid K\}^\omega) \\ dom(O_1) \cap dom(O_2) = \{\} \quad edom(O_1) \cap edom(O_2) = \{\} \quad ((\mathbf{c}_i, c_i) : T_i) \in O_i \\ m = \mid ES_1 \mid_{c_1}, n = \mid ES_2 \mid_{c_2}, j \cdot m = k \cdot n = lcm(m,n) \\ K = (K_1 \cup K_2) \setminus \{c_1, c_2\} \quad O = (O_1 \cup O_2) \setminus \{c_1, c_2\} \quad \pi_1 = +, \pi_2 = - \\ FS = \{((j \cdot ES_1 \uplus k \cdot ES_2) \setminus \{c_1, c_2\}) \mid K\}^\omega \end{array}}{\Gamma, K \vdash_\Sigma \mathsf{connect}_{m,n}(v_1.\mathbf{c}_1, v_2.\mathbf{c}_2) : \mathsf{actsig}(O, FS) : \{\}} \ \textsc{Conn}$$

$$\frac{\begin{array}{c} \Gamma \vdash_\Sigma v : \mathsf{actsig}(O, \{ES \mid K\}^\omega) \quad ((\mathbf{c}_1, c_1) : T_1), ((\mathbf{c}_2, c_2) : T_2) \in O \\ K_0 = K \setminus \{c_1, c_2\} \quad \mid ES \mid_{c_1} = m = \mid ES \mid_{c_2} \quad \pi_1 = +, \pi_2 = - \\ ES_0 = ES \setminus \{c_1, c_2\} \quad O_0 = O \setminus \{c_1, c_2\} \quad K, \Gamma \nvdash_\Sigma c_1 < c_2 \end{array}}{\Gamma, K \vdash_\Sigma \mathsf{connectSelf}_m(v.\mathbf{c}_1, v.\mathbf{c}_2) : \mathsf{actsig}(O_0, \{ES_0 \mid K_0\}^\omega) : \{\}} \ \textsc{Conn Self}$$

$$\frac{\begin{array}{c} \Gamma \vdash_\Sigma v : \mathsf{actsig}(O, \{ES \mid K\}^\omega) \quad ((\mathbf{c}_1, c_1) : T_1), ((\mathbf{c}_2, c_2) : T_2) \in O \\ K_0 = K \setminus \{c_1, c_2\} \quad \mid ES \mid_{c_1} = m = \mid ES \mid_{c_2} \quad \pi_1 = +, \pi_2 = - \\ ES_0 = ES \setminus \{c_1, c_2\} \quad O_0 = O \setminus \{c_1, c_2\} \quad K, \Gamma \vdash_\Sigma c_1 < c_2 \end{array}}{\Gamma, K \vdash_\Sigma \mathsf{connectSelfDelay}_m(v.\mathbf{c}_1, v.\mathbf{c}_2) : \mathsf{actsig}(O_0, \{ES_0 \mid K_0\}^\omega) : \{\}} \ \textsc{Conn Delay}$$

Fig. 8. Actor type rules

The type environment is not treated linearly, since we are not tracking usage of resources. We instead rely on matching a statement against a flowstate during type-checking. We also rely on some other meta-notations:

1. The expression $O \setminus V$ denotes the removal of all bindings for channel names in V from O:
$$O \setminus V = \{((\mathbf{c},c):T) \in O \mid c \notin V\}.$$
We sometimes denote $O \setminus \{c\}$ by $O \setminus c$.
2. For a set of causalities K, we denote the removal of all constraints involving channels in V by $K \setminus V$. In other words, $K \setminus V = \{(c_1 < c_2) \in K \mid V \cap \{c_1, c_2\} = \{\}\}$.
3. For event states, we denote a multiset by the set of elements with their multiplicities, so $ES = \{m_1 \cdot a_1, \ldots, m_k \cdot a_k\}$ contains m_i occurrences of a_i (assuming $a_i \notin \{a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_k\}$). Denote the number of occurrences of a_i in ES by $\mid ES \mid_{a_i} = m_i$, and say that $c \in ES$ if and only if $\mid ES \mid_c > 0$. The disjoint union $ES_1 \uplus ES_2$ adds the multiplicities of common elements, so $\mid ES_1 \uplus ES_2 \mid_c = \mid ES_1 \mid_c + \mid ES_2 \mid_c$. The expression $ES \setminus \{n \cdot c\}$ denotes the removal of n of occurrences of c from the multiset ES, so $\mid ES \setminus \{n \cdot c\} \mid_c = max(\mid ES \mid_c - n, 0)$. The expression $n \cdot ES$ denotes the multiplication of the multiplicities in ES by n: $n \cdot ES = \{n \cdot m \cdot a \mid m \cdot a \in ES\}$.
4. Finally we denote the projection of an event state onto the names that are in a set of variables (typically the domain of a type environment) by:
$$ES[\![V]\!] = \{(m \cdot c) \in ES \mid c \in V\}.$$
The homomorphic extension of this to the projection of a flowstate is denoted by $FS[\![V]\!]$.

$$\pi \leq \pi \quad \pm \leq + \quad \pm \leq -$$

$$\frac{\Gamma \vdash K \ \mathsf{ok} \quad (c_1 < c_2) \in K}{\Gamma, K \vdash_\Sigma c_1 < c_2} \ \text{CAUS HYP}$$

$$\frac{\Gamma, K \vdash_\Sigma c_1 < c_2 \quad \Gamma, K \vdash_\Sigma c_2 < c_3}{\Gamma, K \vdash_\Sigma c_1 < c_3} \ \text{CAUS TRANS}$$

Fig. 9. Subtyping and subflow rules

The type system is formulated using judgements of the following forms:

$\vdash_\Sigma \Gamma \ \mathsf{ok}$	Environment
$\Gamma \vdash_\Sigma T$	Type
$\Gamma \vdash_\Sigma K$	Causal Set
$\Gamma, K \vdash_\Sigma a < b$	Causality
$\Gamma \vdash_\Sigma v : T$	Value
$\Gamma, K \vdash_\Sigma e : T : FS$	Expression
$\Gamma, K \vdash_\Sigma s : FS$	Statement

The main type rules are provided in Fig. 7. The CONST, VAR and NAME rules are used to type check values; variables and names should be bound in the environment Γ. The skip construct has empty effect, while the conditional is required to have the same effect in both branches of the conditional. There are two looping constructs. The default rule, for while loops, has an iteration type FS^*. This may not be sufficient for some circumstances, in particular for the top-level loop of an actor that is required to always offer the specified behavior (after a complete firing). Therefore we provide an additional loop construct, to separate loops in the type system that are guaranteed to never terminate. While our type system sometimes requires infinite loops, it does not attempt to prevent infinite loops, so it is possible for an actor network to fail to make observable progress because an actor is stuck in an internal loop. The progress result for the operational semantics guarantees that the network can always make progress, even if this progress is just the iteration of an infinite loop without observable behavior.

For tracking flowstate, the key rule is the BIND rule, which evaluates an expression e, and binds a new variable x to the result of this evaluation when evaluating the statement s. Since the expression may include message sending or receiving, it has a flowstate that is combined with the flowstate of the statement continuation.

The various forms of expressions are typed by the remaining rules in Fig. 7. The BUILTIN rule type-checks a arithmetic expression resulting from the application of a built-in function. The ASSIGN rule type-checks an assignment expression, which again is restricted to primitive values of base type (i.e., float). The SEND and RECEIVE rules type-check the message sending and receiving operations. The channel must have the appropriate polarity in the environment, - for sending and + for receiving. The RUN rule runs a dataflow network for which all open channels have been resolved. The return value is a dummy value. The dataflow network operates asynchronously with the parent network.

The STMT EQ rule allows the flowstate for a statement to be replaced with a flowstate that is provably equal to it. We define the notion of equality between flowstates in the next section, defining it as a bisimulation between flowstate computations.

The difficult rules are those for connecting actors together into composite actors. These rules are provided in Fig. 8, where the rules for actor expressions are provided. The ACTOR rule type checks an atomic actor expression, checking the body of the actor in an environment binding the open channels with the appropriate polarities. The CONN rule handles the case where two different actors are being connected. As we have seen, it is because of this rule, and the difficulties with aliasing, that we do not allow actor references to be copied in this language. The rule requires that the open channel names in the two actors are distinct. In practice it would obviously be useful to have a way to rename these when necessary, but it is not essential for the current account. The top-level flowstate is required to be an infinite loop type for each actor, and the new flowstate results from a merging of these infinite loop types, removing all references to the open channels on which the actors are being linked. These channel names are also removed from the open channel set for the resulting combination.

The CONN SELF and CONN DELAY rules handle the linking of two open channels in the same actor. The first of these handles the case where the addition of the new binding does not introduce a feedback loop, as reflected by the causality rules that require the data paths in elided channels within the actor. The second of these two rules handles the case where the linking would result in a feedback loop, and therefore fills the buffer for the private channel linking the open channels to introduce a delay in firing.

Our language is essentially synchronous data flow, however its generalization to cyclostatic is straightforward. We restrict the flowstate of an actor to have the form $\{FS \mid K\}^{\omega}$. To extend this to cyclostatic behavior, we generalize the form of the body $(\{FS_1 \mid K_1\};\ldots;\{FS_m \mid K_m\})^{\omega}$. The main complication is in the type rules for connecting actors, and we eschew the details in this account.

5 Semantics

In this section, we consider an operational semantics for the language described in the previous section. We provide a heap-based semantics that binds three kinds of values on the heap:

1. Values n of base type, i.e., of type float. Every variable of type float is assumed to be mutable, therefore we bind such variables to locations l that point to their heap binding.
2. Actor values A, which may be either atomic or composite, resulting from allocating atomic actors and then connecting them together on open channels. We extend actors with *shared channel bindings* S, containing bindings of the form $(c : (k, m, T))$. An atomic actor is a special case of a composite actor:

$$\mathsf{actor}(O, s) \equiv \mathsf{actor}(O, \{\}, s)$$

   That is, an atomic actor has no shared channels, only open channels, and a single thread.

$$
\begin{array}{rcl}
T \in \text{Type} & ::= & [T]_k \\
S \in \text{Shared Channels} & ::= & \{\} \mid \{(c:(k,n,T))\} \mid S \cup S \\
H \in \text{Heap} & ::= & \varepsilon \mid l \mapsto n \mid a \mapsto A \mid c \mapsto B \mid H_1 \uplus H_2 \\
HT \in \text{Heap Type} & ::= & \varepsilon \mid l : \mathsf{float} \mid a : AS \mid c : \mathsf{channel}\ \pi \mid HT_1 \uplus HT_2 \\
A \in \text{Actor} & ::= & \mathsf{actor}(O,S,P) \\
P \in \text{Process} & ::= & \mathsf{stop} \mid FS \mid (P_1 \mid P_2) \\
B \in \text{Buffer} & ::= & \varepsilon_k \mid [v]_k \mid B_1 @_k B_2 \\
C \in \text{Configuration} & ::= & (e,H) \mid (s,H) \mid (P,H)
\end{array}
$$

Fig. 10. Syntax of configurations

3. Message buffers B, which hold the values transmitted between actors on shared channels. A message buffer is simply a sequence, ensuring FIFO delivery, where $_ @_k _$ is the operation for appending buffers. We assume that buffers have bounded size, provided by a parameter k in the constructors and in the buffer type; the constructor operations are undefined for the case where the resulting buffer is larger than the maximum size. We denote the number of items in a buffer by $\mid B \mid$, and the maximum size of a buffer by $size(B)$. We write $[v_1, v_2, \ldots, v_m]_k$ as an abbreviation for $[v_1]_k @_k [v_2]_k @_k \ldots @_k [v_m]_k$, where $m \leq k$. We use $v ::_k B$ to denote $[v]_k @_k B$. We use $[T]_k$ to denote the type of a buffer that contains values of type T. These buffer types are not first class, since buffers are handled by the compiler. Furthermore for simplicity we restrict the contents of buffers to be floating point values, so $T = \mathsf{float}$ for any buffer type $[T]_k$.

To describe the operational semantics, we generalize the form of an actor, as described in Fig. 10. An open channel always has polarity of + or −, reflecting the fact that it is a uniplex channel. When two such shared channels are bound to the endpoints of a shared channel, the latter has polarity $\pm$. For open channels, we use the channel names as representatives of events, since an open channel is either input or output, but not both. When we instantiate open channels to shared channels, we need to distinguish the sending and receiving ends of the shared channel, for dependency checking purposes, since we need to distinguish sending and receiving events on that channel. We assume a naming convention where, for any shared channel c, there are distinguished names c^+ and c^- for the receiving and sending parts, respectively, of that channel. The receiving end c^+ of a shared channel has polarity +, while the sending end c^- has polarity − (while the channel c itself has polarity $\pm$). Whereas open channels have environment bindings of the form $(c : \mathsf{channel}\ +)$ and $(c : \mathsf{channel}\ -)$, a shared channel binding has the form $(c : (k, n, \mathsf{channel}\ \pm))$, that also implicitly binds the names c^+ and c^- for the two ends of the channel. The parameter k denotes the maximum size required of the corresponding buffer, while the parameter n denotes that a delay must be introduced in the channel for a buffer, by initializing that buffer with n default values (either $n = k$ or $n = 0$). We denote the operation of adjusting these parameters, as a result of connecting two actors and adjusting their firing rates, by:

$$j \cdot S = \{(c : (j \cdot k, j \cdot n, T)) \mid (c : (k, n, T)) \in S\}.$$

In addition to the set of open channels O in an actor interface, we now also have a set of shared channels S. We associate with each shared channel in S a number that is the number of initial values to be inserted into a buffer when it is created. When this value is non-zero, the buffer is required to provide a delay in the firing semantics.

The other extension to an actor expression is that the body is generalized from a single thread to multiple threads, arising from the linking of actors together into a composite actor or dataflow network. The body of a composite actor is the parallel composition of a collection of single-threaded actor bodies, where each body has the flowstate specification given by the original atomic actor expression. The one change is that open channels in the original flowstate constraint will have been replaced by an endpoint of a shared channel, as a result of connecting that actor with another actor. By the time a dataflow network runs, all channel names in flowstates will have been replaced by shared channel endpoints. A configuration of the operational semantics is simply a parallel composition of threads paired with a global heap. This heap is actually partitioned between the different dataflow nets that are running.

In order to reason about correctness, we define typing relations for heaps and processes, using judgements of the form:

$$\begin{array}{ll} \Gamma \vdash_{\Sigma} H : HT & \text{Heap} \\ \Gamma \vdash_{\Sigma} P : FS & \text{Process} \\ \Gamma \vdash_{\Sigma} B : [T]_k & \text{Buffer} \\ \Gamma \vdash_{\Sigma} A : AS & \text{Actor} \end{array}$$

This last judgement just checks for well-formedness of the channel flow constraints in an actor. The channels named in the constraints should be bound in the type environment (obtained from the open channels in the case of an actor signature, and from the open and shared channels in the case of an actor expression). The type rules are provided in Fig. 11. In the ACTOR rule for typing actor bodies, the closure operation $\mathscr{C}(K)$ forms the transitive closure of a set of causality constraints.

For evaluating expressions, mutable base type variables are bound to locations l, and these must be dereferenced. This dereferencing is performed by the operation of applying the heap to a value, $H(v)$, defined by:

$$\begin{aligned} H(l) &= n \text{ if } l \mapsto n \in H \\ H(a) &= A \text{ if } a \mapsto A \in H \\ H(c) &= B \text{ if } c \mapsto B \in H \\ H(v) &= v \text{ otherwise} \end{aligned}$$

For built-in functions, we assume a family of total functions $\{eval_f\}_f$ for the functions defined in the signature Σ.

$$\frac{\vdash_\Sigma \Gamma \text{ ok}}{\Gamma \vdash_\Sigma \mathsf{stop} : \{\}} \text{ (Proc Stop)}$$

$$\frac{\Gamma, \{\} \vdash_\Sigma s : FS}{\Gamma \vdash_\Sigma s : FS} \text{ (Proc Stmt)}$$

$$\frac{\Gamma \vdash_\Sigma P_1 : FS_1 \quad \Gamma \vdash_\Sigma P_2 : FS_2}{\Gamma \vdash_\Sigma (P_1 \mid P_2) : FS_1 \parallel FS_2} \text{ (Proc Par)}$$

$$\frac{K' \supseteq K \cap fn(FS) \quad \Gamma \vdash_\Sigma P : \{FS \mid K\}}{\Gamma \vdash_\Sigma P : \{FS \mid K\}} \text{ (Proc Cause)}$$

$$\frac{\vdash_\Sigma \Gamma \text{ ok}}{\Gamma \vdash_\Sigma \varepsilon_k : [T]_k} \text{ (Buff Empty)}$$

$$\frac{\Gamma \vdash_\Sigma v : T}{\Gamma \vdash_\Sigma [v]_k : [T]_k} \text{ (Buff Cell)}$$

$$\frac{\Gamma \vdash_\Sigma B_1 : [T]_k \quad \Gamma \vdash_\Sigma B_2 : [T]_k}{\Gamma \vdash_\Sigma B_1 @_k B_2 : [T]_k} \text{ (Buff Join)}$$

$$\frac{\vdash_\Sigma \Gamma \text{ ok}}{\Gamma \vdash_\Sigma \varepsilon : \varepsilon} \text{ (Heap Empty)}$$

$$\frac{\Gamma \vdash_\Sigma n : \mathsf{float}}{\Gamma \vdash_\Sigma (l \mapsto n) : (l : \mathsf{float})} \text{ (Heap Float)}$$

$$\frac{T = \mathsf{float} \quad \Gamma \vdash_\Sigma B : [T]_k}{\Gamma \vdash_\Sigma (c \mapsto B) : (c : [T]_k)} \text{ (Heap Buffer)}$$

$$\frac{\Gamma \vdash_\Sigma A : AS}{\Gamma \vdash_\Sigma (a \mapsto A) : (a : AS)} \text{ (Heap Actor)}$$

$$\frac{\Gamma \vdash_\Sigma H_1 : HT_1 \quad \Gamma \vdash_\Sigma H_2 : HT_2}{\Gamma \vdash_\Sigma H_1 \uplus H_2 : HT_1 \uplus HT_2} \text{ (Heap Join)}$$

$$\frac{\begin{array}{c}\Gamma_0 = \{(c : T) \mid ((\mathbf{c}, c) : T) \in O\} \quad \Gamma_1 = \{(c : T) \mid (c : (k, n, T)) \in S\} \\ dom(\Gamma_0) \cap dom(\Gamma_1) = \{\} \quad \Gamma_0 \cup \Gamma_1 \vdash_\Sigma P : FS \quad FS_0 = FS[\![dom(\Gamma_0)]\!]\end{array}}{\Gamma \vdash_\Sigma \mathsf{actor}(O, S, P) : \mathsf{actsig}(O, FS_0)} \text{ (Actor)}$$

Fig. 11. Type Rules for Heaps and Processes

$$(\mathsf{stop} \mid P) \equiv P \qquad (P_1 \mid P_2) \equiv (P_2 \mid P_1) \qquad (P_1 \mid (P_2 \mid P_3)) \equiv ((P_1 \mid P_2) \mid P_3)$$

$$(\mathsf{skip}; s) \equiv s \qquad (s; \mathsf{skip}) \equiv s \qquad (s_1; (s_2; s_3)) \equiv ((s_1; s_2); s_3)$$

Fig. 12. Structural equivalence

The semantics is defined using a collection of reduction relations:

Reduction of expressions: $(e_1, H_1) \longrightarrow (e_2, H_2)$ and $(e_1, H_1) \xrightarrow{a} (e_2, H_2)$
Reduction of statements: $(s_1, H_1) \longrightarrow (s_2, H_2)$ and $(s_1, H_1) \xrightarrow{a} (s_2, H_2)$
Reduction of processes: $(P_1, H_1) \longrightarrow (P_2, H_2)$ and $(P_1, H_1) \xrightarrow{a} (P_2, H_2)$
Reduction of types: $K \vdash FS_1 \xrightarrow{a} FS_2$

$$\frac{v = eval_f(H(v_1),\ldots,H(v_k))}{(f(v_1,\ldots,v_k),H) \longrightarrow (v,H)}\ \textsc{Builtin} \qquad \frac{n = H(v) \quad H' = H[l \mapsto n]}{(l = v,H) \longrightarrow (n,H')}\ \textsc{Assign}$$

$$\frac{n = H(v) \quad k = size(H(c)) \quad |\,H(c)\,| < k \quad H' = H[c \mapsto H(c)@_k[n]_k]}{(c^- \uparrow v,H) \xrightarrow{c^-} (0,H')}\ \textsc{Send}$$

$$\frac{H(c) = [n]_k @_k B \quad H' = H[c \mapsto B]}{(c^+ \downarrow,H) \xrightarrow{c^+} (n,H')}\ \textsc{Receive}$$

$$\frac{(e,H) \xrightarrow{[a]} (n,H') \quad l \notin dom(H) \quad H' = H \cup \{l \mapsto n\}}{((\mathsf{var}\ x = e;s),H) \xrightarrow{[a]} (\{l/x\}s,H')}\ \textsc{Bind}$$

$$\frac{H(v) \neq 0}{((\mathsf{if}\ (v)\ s_1;\ \mathsf{else}\ s_2),H) \longrightarrow (s_1,H)}\ \textsc{If True} \qquad \frac{H(v) = 0}{((\mathsf{if}\ (v)\ s_1;\ \mathsf{else}\ s_2),H) \longrightarrow (s_2,H)}\ \textsc{If False}$$

$$\frac{(s,H) \xrightarrow{a} (s',H)}{((\mathsf{fire}_K\ s),H) \xrightarrow{a} ((\mathsf{fire}_K\ s'),H)}\ \textsc{Fire}$$

$$\frac{H(v) \neq 0}{((\mathsf{while}\ (v)\ s),H) \longrightarrow ((s;\mathsf{while}\ (v)\ s),H)}\ \textsc{While True}$$

$$\frac{H(v) = 0}{((\mathsf{while}\ (v)\ s),H) \longrightarrow (\mathsf{skip},H)}\ \textsc{While False}$$

$$\frac{(s_1,H) \xrightarrow{[a]} (s_1',H')}{((s_1;s_2),H) \xrightarrow{[a]} ((s_1';s_2),H')}\ \textsc{Seq} \qquad \frac{(P_1,H) \xrightarrow{[a]} (P_1',H')}{((P_1 \mid P_2),H) \xrightarrow{[a]} ((P_1' \mid P_2),H')}\ \textsc{Par}$$

$$\frac{\begin{array}{c} H(v) = \mathsf{actor}(\{\},S,P) \quad dom(H) \cap \{c \mid (c:(n,T,\in))S\} = \{\} \\ H_2 = H_1 \cup \{c \mapsto dbuf_k(n) \mid (c:(n,T,\in))S\} \end{array}}{((\mathsf{run}\ v;s),H_1) \longrightarrow ((s \mid P),H_2)}\ \textsc{Run}$$

Fig. 13. Operational Semantics

The first three pairs relations describe reductions between configurations of an expression, statement and process, respectively, coupled with a heap. The heap is both input into, and output from, each reduction step. A reduction of expressions of the form $(e_1,H_1) \longrightarrow (e_2,H_2)$ denotes an internal reduction, while a reduction of the form $(e_1,H_1) \xrightarrow{a} (e_2,H_2)$ denotes a reduction that involves a communication event a. We write $(e_1,H_1) \xrightarrow{[a]} (e_2,H_2)$ to generically denote a reduction that may be either internal or involve a communication event. Similar remarks hold for reduction of statements and of processes.

$$\frac{a \notin dom(H) \quad e = \mathsf{actor}(O,s) \quad H' = H \cup \{a \mapsto \mathsf{actor}(O,\{\},s)\}}{((\mathsf{var}\ x = e; s), H) \longrightarrow (\{a/x\}s, H')} \text{ ACTOR}$$

$$\frac{\begin{array}{c} H(a_i) = \mathsf{actor}(O_i,S_i,P_i) \quad ((\mathbf{c}_i,c_i):T_i) \in O_i \quad O = (O_1 \cup O_2) \setminus \{c_1,c_2\} \\ \pi_1 = +, \pi_2 = - \quad j_1 \cdot m = j_2 \cdot n = lcm(m,n) = k \quad dom(S_1) \cap dom(S_2) = \{\} \\ c \notin dom(S_1) \cup dom(S_2) \quad S = j_1 \cdot S_1 \cup j_2 \cdot S_2 \cup \{(c:(k,0,T))\} \\ P_1' = \{c^+/c_1\}P_1 \quad P_2' = \{c^-/c_2\}P_2 \quad A = \mathsf{actor}(O,S,(P_1' \mid P_2')) \end{array}}{((\mathsf{var}\ x = \mathsf{connect}_{m,n}(a_1.c_1,a_2.c_2);s),H) \longrightarrow (\{a/x\}s, H \cup \{a \mapsto A\})} \text{ CONN}$$

$$\frac{\begin{array}{c} H(a) = \mathsf{actor}(O,S,P) \quad ((\mathbf{c}_1,c_1):T_1),((\mathbf{c}_2,c_2):T_2) \in O \\ dom(S_1) \cap dom(S_2) = \{\} \quad c \notin dom(S_1) \cup dom(S_2) \\ S' = S \cup \{(c:(m,m,T))\} \quad \pi_1 = +, \pi_2 = - \quad O' = O \setminus \{c_1,c_2\} \\ P' = \{c^+/c_1, c^-/c_2\}P \quad A = \mathsf{actor}(O',S',P') \end{array}}{((\mathsf{var}\ x = \mathsf{connectSelfDelay}_m(a.c_1,a.c_2);s),H) \longrightarrow (\{a/x\}s, H \cup \{a \mapsto A\})} \text{ CONN DELAY}$$

Fig. 14. Actor operational semantics

The reduction relation for flowstates is perhaps surprising, and reflects the use of flowstate: Types themselves evolve during computation, since they are abstract process descriptions for the underlying sequential program. This reduction relation is the basis for a type equality for types, based on bisimulation:

Definition 1 (Flowstate Equality). *Given a causality set K. Define K-*bisimilarity *to be the largest symmetric binary relation R defined by: If $(FS_1, FS_2) \in R$, then if $K \vdash FS_1 \xrightarrow{a} FS_1'$ for some FS_1', then $K \vdash FS_2 \xrightarrow{a} FS_2'$ for some FS_2' such that $(FS_1', FS_2') \in R$. If FS_1 and FS_2 are K-bisimilar, then we denote this by $\Gamma, K \vdash_\Sigma FS_1 \cong FS_2$.*

The RUN rule for launching a dataflow net initializes buffers using the function $dbuf_k(n)$, defined by:

$$dbuf_k(0) = \varepsilon_k$$
$$dbuf_k(n+1) = 0 ::_k dbuf_k(n)$$

The operational semantics for agent expressions are provided in Fig. 14. The reduction relation for flowstates is defined in Fig. 15.

The formal results for the type system are in two parts:

1. *Type preservation* verifies that the well-typedness (but not the types!) of actors are preserved under evaluation.
2. *Progress* verifies that, given a flowstate that can perform a reduction step, a corresponding actor with that flowstate either diverges (loops infinitely) or eventually (after internal reductions) can simulate that abstract reduction step.

Theorem 1 (Type Preservation). *If $\Gamma, K \vdash_\Sigma (P_1, H_1) : FS_1$ and $(P_1, H_1) \xrightarrow{a} (P_2, H_2)$ then $K \vdash FS_1 \xrightarrow{a} FS_2$, and $\Gamma, K \vdash_\Sigma (P_2, H_2) : FS_2$.*

$$\{\}^* \equiv \{\} \qquad FS^\omega \equiv FS;FS^* \qquad FS_1 \parallel FS_2 \equiv FS_2 \parallel FS_1$$

$$(FS_1;FS_2);FS_3 \equiv FS_1;(FS_2;FS_3) \qquad (FS_1 \parallel FS_2) \parallel FS_3 \equiv FS_1 \parallel (FS_2 \parallel FS_3)$$

$$FS_1^\omega \parallel FS_2^\omega \equiv (FS_1 \parallel FS_2)^\omega$$

$$\{FS_1 \mid K\} \parallel \{FS_1 \mid K\} \equiv \{(FS_1 \parallel FS_2) \mid K\}$$

$$\frac{K \vdash FS_1 \xrightarrow{a} FS_1'}{K \vdash (FS_1 \parallel FS_2) \xrightarrow{a} (FS_1' \parallel FS_2)}$$

$$\frac{K_0 \cup K \vdash FS \xrightarrow{a} FS'}{K_0 \vdash \{FS \mid K\} \xrightarrow{a} \{FS \mid K\}}$$

$$\frac{K \vdash FS_1 \xrightarrow{a} FS_1'}{K \vdash (FS_1;FS_2) \xrightarrow{a} (FS_1';FS_2)}$$

$$\frac{K \vdash FS \xrightarrow{a} FS'}{K \vdash (\{\};FS) \xrightarrow{a} (\{\};FS)}$$

$$\frac{m > 0 \qquad \nexists a_0 \in ES.\Gamma, K \vdash_\Sigma a_0 < a}{K \vdash \{m \cdot a\} \uplus ES \xrightarrow{a} \{(m-1) \cdot a\} \uplus ES}$$

$$\frac{FS_1 \equiv FS_1' \qquad K \vdash FS_1' \xrightarrow{a} FS_2' \qquad FS_2 \equiv FS_2'}{K \vdash FS_1 \xrightarrow{a} FS_2}$$

Fig. 15. Type reduction rules

Let $(P,H) \Longrightarrow (P',H')$ denote the reflexive transitive closure of $(P,H) \longrightarrow (P',H')$: in other words, (P,H) evolves to (P',H') in zero or more internal reductions. Let $(P_1,H_1) \stackrel{a}{\Longrightarrow} (P_2,H_2)$ denote that $(P_1,H_1) \Longrightarrow (P_1',H_1')$ and $(P_1',H_1') \stackrel{a}{\longrightarrow} (P_2',H_2')$ and $(P_2',H_2') \Longrightarrow (P_2,H_2)$. Let $(P_1,H_1) \xRightarrow{(a_1,\ldots,a_k)} (P_{k+1},H_{k+1})$ denote that $(P_i,H_i) \stackrel{a_i}{\Longrightarrow} (P_{i+1},H_{i+1})$ for $i = 1,\ldots,k$ and some $(P_1,H_1),\ldots,(P_{k+1},H_{k+1})$ and $a_1,\ldots,a_k$.

Denote that a configuration (P,H) *diverges*, in the sense that it loops indefinitely performing only internal reductions, by $(P,H)\Uparrow$. Denote that a configuration eventually offers output on channel endpoint c^-, perhaps after performing some internal reductions, by $(P,H) \Downarrow_{c^-}$. Similarly $(P,H) \Downarrow_{c^+}$ denotes that a configuration eventually attempts to perform input on channel endpoint c^+, perhaps after performing some internal reductions.

Theorem 2 (Progress). *If* $\Gamma, K \vdash_\Sigma (P,H) : FS_1$ *and* $\left\{\begin{array}{l} K \vdash FS_1 \xrightarrow{c^+} FS_2 \\ K \vdash FS_1 \xrightarrow{c^-} FS_2 \end{array}\right\}$ *then either* $(P,H)\Uparrow$*, or* $\left\{\begin{array}{l} (P,H) \xRightarrow{\overrightarrow{a^+}} (P',H') \\ (P,H) \xRightarrow{\overrightarrow{a^-}} (P',H') \end{array}\right\}$ *for some* (P',H') *and* $\overrightarrow{a}$ *such that* $\left\{\begin{array}{l} (P',H') \Downarrow_{c^+} \\ (P',H') \Downarrow_{c^-} \end{array}\right\}$.

6 Related Work

The area of synchronous dataflow has seen some application in signal processing applications [13], with various extensions, in particular cyclostatic dataflow for actors whose firing behavior is able to evolve in a regular fashion [2]. Traditionally the analysis of synchronous dataflow has been non-modular, requiring analysis of the entire dataflow graph. More recent work has considered the modular composition of hierarchical SDF graphs [17], allowing graphs to be analyzed before the entire dataflow graph is constructed, with a focus on modular code generation. The interface of a modular actor is described as a deterministic SDF with shared FIFOs (DSSF) profile, allowing a collection of actors to share an input queue while retaining determinacy of the execution. As with SDF, atomic actors are considered as "black boxes," and only SDF is considered. In particular cyclostatic dataflow is not considered in that work.

Sessional dataflow comes out of the realm of linear [11,18] and affine type systems for statically checking the safe usage of limited resources. Two particularly significant lines of study in the "linear types" field have been the approach of *typestate* and that of *session types*. Typestate is a concept that originated in the Hermes language of Strom and Yemini [15]. It corresponds to an enrichment of the normal notion of a type, to include the concept of types as states in a finite state machine. Fähndrich and Deline incorporated this idea into object oriented languages [7] in a very natural way: each object has a typestate, and the interface offered by an object, in the sense of the methods that can currently be invoked on the object, are determined by its current typestate. Since typestate is updated imperatively, it is important that aliasing of such objects be carefully controlled. Aldrich et al [14] have demonstrated that a notion of *permissions*, based on earlier work on type-based capabilities, can be used to check the use of typestate in existing non-toy software systems. We have explicitly avoided introducing these issues into the current report, but they are clearly relevant to incorporating sessional dataflow into real programming languages.

The approach of session types [9] is commonly motivated by its support for safe Web services. In the simplest case, session types are used to mediate the exchanges between two parties in a dyadic interaction. Each session offers a "shared channel" (different from our use of the terminology), essentially an service endpoint URL that a client connects to. On connection, a new server thread is forked and a private session channel is established between the client and this thread. This channel has a behavioral type that is essentially an abstract single-threaded process, that constrains the communications between the parties. Since only the client and the server share their private channel, the execution is in fact deterministic.

Although sessional dataflow might appear at first related to session types, the connection is actually rather weak, because of the nature of the interactions in dataflow. The closest our system comes to a session types system is in the behavioral constraint on the behavior of an actor, in terms of matching the specified input and output data rates on each firing specified on an actor interface. However this behaviorial specification only constrains a single actor, and places no constraint on the behavior of its neighboring actors (upstream or downstream). Furthermore a session type specifies, for each participant in an interaction, a very precise single-threaded behavior, in terms of data

exchanged on the private session channels at each point in the execution. In contrast, the behavioral specification for an actor in sessional dataflow is declarative, specifying expected communications subject to causality constraints. Deniélou and Yoshida [8] describe a version of session types that allows a dynamic number of participants in a session protocol. As with other approaches to session types, the approach is to provide operational specifications of participant behaviors, using a top-down approach where one reasons from the specified global protocol to the behavior of individual participants. In contrast, the sessional dataflow approach is bottom-up and declarative, specifying declarative causality constraints on individual actors independent of whatever interactions they are integrated into.

Another related line of work is in synchronous languages for real-time and embedded systems. Such languages assume a "clock" on all computations, with variables representing potentially infinite streams of values, indexed by clock ticks. Here the most relevant example for sessional dataflow is that of Lustre [4], a language that is a dataflow language in the tradition of Lucid, [1], and is a synchronous language in the sense of the synchronous languages such as Esterel [3], but which we cannot call a synchronous dataflow language for fear of confusing the reader. The constraints on the synchronous languages preclude any need for buffering, since all actors operate in lock step on the same clock. The theory of these "synchronous," "dataflow" networks has been described in terms of synchronous Kahn networks [5], which have the property that no buffering is required at all between actors, since all execution is synchronous and governed by a common clock. This is clearly a very strong restriction, albeit one that facilitates compilation of programs to hardware circuits. The theory of N-synchronous Kahn networks [6] relaxes this restriction, allowing different actors to have their own clock rates, and allowing buffering between actors to match their clock rates. It is therefore very much related to the approach of synchronous dataflow, with subtyping between clock rates in multi-rate systems identifying where data must be buffered. However matching data rates does not address the other aspect of sessional dataflow, causalities to ensure the liveness of networks as they are composed.

7 Conclusions

This work builds on existing work in dataflow computation, particularly the work in synchronous dataflow pursued in the signal processing community, as discussed in Sect. 1. Our work considers how to relate the implementations of actors to the static firing rates described in actor interfaces, where the latter are critical for static scheduling of actors. We have also provided a compositional semantics for combining actors together into dataflow nets, in such a way that we statically check the correctness of the combination at each step of such a process.

There are extensions of synchronous dataflow that can be incorporated into sessional dataflow, such as the extension to cyclostatic dataflow considered at the end of Sect. 4. However our main interest is in using the framework of sessional dataflow to consider the safety of operations such as reconfiguration and subnet replacement.

References

1. Ashcroft, E.A., Wadge, W.W.: Lucid, the dataflow programming language. Academic Press (1985)
2. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.A.: Cyclo-static data flow. In: International Conference on Acoustics, Speech, and Signal Processing (ICASSP), vol. 5, pp. 3255–3258 (May 1995)
3. Boussinot, F., de Simone, R.: The Esterel language. Proc. IEEE 79, 1270–1282 (1991)
4. Boussinot, F., de Simone, R.: The synchronous data flow programming language Lustre. Proc. IEEE 79, 1305–1320 (1991)
5. Caspi, P., Pouzet, M.: Synchronous kahn networks. In: International Conference on Functional Programming, ICFP (1996)
6. Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., Pouzet, M.: N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. In: Principles of Programming Languages (POPL), pp. 180–193. ACM Press (2006)
7. DeLine, R., Fähndrich, M.: Typestates for Objects. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 465–490. Springer, Heidelberg (2004)
8. Deniélou, P.-M., Yoshida, N.: Dynamic multirole session types. In: ACM Symposium on Principles of Programming Languages, pp. 435–446. ACM, New York (2011)
9. Dezani-Ciancaglini, M., de'Liguoro, U.: Sessions and Session Types: An Overview. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 1–28. Springer, Heidelberg (2010)
10. Edwards, S.A.: Languages for Digital Embedded Systems. Kluwer (2000)
11. Girard, J.-Y.: Linear logic. Theoretical Computer Science (50), 1–102 (1987)
12. Kahn, G.: The semantics of a simple language for parallel programming. In: Information Processing 74: Proceedings of the IFIP Congress, pp. 471–475. North-Holland, Stockholm (1974)
13. Lee, E., Messerschmitt, D.: Synchronous data flow. Proc. IEEE 75(9), 1235–1245 (1987)
14. Stork, S., Marques, P., Aldrich, J.: Concurrency by default: using permissions to express dataflow in stateful programs. In: Proceeding of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, OOPSLA 2009, pp. 933–940. ACM, New York (2009)
15. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. 12, 157–171 (1986)
16. Thies, W.: Language and Compiler Support for Stream Programs. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA (February 2009)
17. Tripakis, S., Bui, D., Rodiers, B., Lee, E.A.: Compositionality in synchronous data flow: Modular code generation from SDF graphs. Technical Report UCB/EECS-2009-143, University of California, Berkeley (October 2009)
18. Wadler, P.: Linear types can change the world? In: Programming Concepts and Methods. North (1990)