Processes and Actors: Translating Kahn Processes to Dataflow with Firing

Gustav Cedersjö
Department of Computer Science
Lund University, Sweden
Email: gustav@cs.lth.se

Jörn W. Janneck
Department of Computer Science
Lund University, Sweden
Email: jwj@cs.lth.se

***Abstract*—Dataflow programming is a paradigm for describing stream processing algorithms in a manner that naturally exposes their concurrency and makes the resulting programs readily implementable on highly parallel architectures.**

Dataflow programs are graph structured, with nodes representing computational kernels that process the data flowing over the edges. There are two major families of languages for the kernels: process languages and languages for dataflow with firing. While processes tend to be easier to write, the additional structure provided by the dataflow-with-firing style increases the analyzability of dataflow programs and supports more efficient implementation techniques.

This paper seeks to combine these benefits in a principled manner by constructing a family of translations from a process language to dataflow with firing. In order to formally relate these descriptions, we first introduce a notion of firing to the semantics of Kahn processes, which allows us to give a precise definition of equivalence between programs written in these different styles. Then we introduce a family of translations between them and and show that they retain meaning of a program. The presented language and its translation has been implemented in a compiler for the dataflow programming language CAL.

I. Introduction

Dataflow programming is a graph-based programming model, where the nodes perform computation on the data that flow over the edges. In the dataflow models we use in this paper, the edges represent buffered, lossless and order-preserving channels. Nodes may have local state variables that are updated throughout the execution, but there are no mutable state variables that are shared between nodes. All communication is done by sending data items (*tokens*) over the channels.

Dataflow programs exhibit a lot of concurrency, because each node can execute independently of the other nodes whenever it has data to process. It also tends to create small modules with few dependencies, which is good for modularity.

There are two major families of languages for expressing the computation in the nodes—process languages and languages for dataflow with firing. Process languages, such as that of Kahn [1], describe the computation in the nodes as sequential programs that explicitly read and write on the channels. Languages for dataflow with firing are instead structured around the concept of a firing, typically describing a set of actions that can be fired upon given conditions, where each action has a known number of tokens it consumes and produces.

```
process SumN() X, N ⟹ Sum :
  n; sum; x;

  repeat
    N ⟶ n;
    sum := 0;
    while n > 0 do
      X ⟶ x;
      sum := sum + x;
      n := n - 1;
    end
    Sum ⟵ sum;
  end
end
```

Fig. 1. A process that computes the sum of n tokens.

The difference in how the languages are structured affects how common idioms are expressed—where process languages can use if and while statements to control the execution, languages for dataflow with firing use the firing conditions of the actions to achieve the same control flow.

Figure 1 and 2 show a program that computes the sum of n tokens, both as a Kahn process and as dataflow with firing. The process version is arguably much simpler, because the control flow better follows the text of the program.

Dataflow programming languages are often associated with a particular execution model or a few different models. Kahn processes, for example, are typically executed using threads or with demand-driven cooperative scheduling, as described in [2], and this paper introduces another execution model for Kahn processes. For dataflow with firing, there are several execution models that take advantage of the firings to create an efficient implementation.

Synchronous dataflow [3] is a model where the number of tokens an actor consumes and produces is the same in every firing. For programs written in this model and for a slightly more general model called cyclo-static dataflow [4], a schedule can be completely determined at compile-time, removing the need for scheduling decisions at runtime. However, not all parts of a program need to be cyclo-static or synchronous dataflow to take advantage of these implementation techniques, as demonstrated in [5] for StreamIt, and in [6] for CAL. Also, [7] and [8] show in two different ways that actors with dynamic token rates can be composed, effectively creating a semi-static

```
actor SumN() X, N ⟹ Sum :
  n; sum;

  start: action N:[nbr] ⟹
  do
    sum := 0;
    n := nbr;
  end

  add: action X:[x] ⟹
  guard n > 0
  do
    sum := sum + x;
    n := n - 1;
  end

  done: action ⟹ Sum:[sum]
  guard n <= 0 end

  schedule Start:
    Start (start) ⟶ Sum;
    Sum (add) ⟶ Sum;
    Sum (done) ⟶ Start;
  end
end
```

Fig. 2. An actor that computes the sum of n tokens.

schedule of the composed actors. Section VI-A discusses a few implementations of dataflow with firing, comparing them to a traditional process implementation.

There are also other benefits of the firing semantics. One is the possibility to record traces of the action firings and create a dependency graph between the firings of a particular execution. In [9], such traces are used to guide the choice of implementation parameters in a design space exploration of a dataflow program.

In this paper we combine the simplicity of processes with the benefits of having them represented as dataflow with firing by designing a Kahn process language with a translation to dataflow with firing. The main contributions of this paper are the translation from the Kahn process language to dataflow with firing, and a way of expressing action firings in the denotational semantics of Kahn processes. Additionally, we elaborate on how the Kahn process source program can be transformed to the recursive functions of its denotational semantics—a detail that is only sketched by Kahn [1].

This paper continues in section II and III with some background on Kahn processes, the process model that we have chosen to implement, and then a short introduction to CAL, the target language of the translation. Section IV introduces a process language whose translation to CAL is presented in section V. Section VI discusses the language design and the translation to dataflow with firing. Related work is discussed in section VII, and finally, section VIII concludes the paper.

II. Process Model

The process model we use in this paper is the one of Kahn [1], often referred to as Kahn process networks. A process is described as a sequential program that can communicate with other processes via blocking reads and non-blocking writes on channels. Kahn showed that a network of such processes always produces the same values on the channels, irrespective of how their executions are interleaved.

A. Semantics

The semantics of Kahn processes have been described in [1], and its details are beyond the scope of this paper. We will, however, discuss some of its building blocks to show the correctness of the translations we are presenting in this paper.

The semantics is denotational rather than operational, and the processes are described as functions on sequences of values. The sequences may be of finite or denumerably infinite length. There is a complete partial order on sequences called the prefix order $\sqsubseteq$, with $a \sqsubseteq b$ if and only if a is the initial segment of b. The functions that describe the processes must be monotonic on this partial order, meaning $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$. Another way to describe prefix monotonicity is that if such a function applied to sequence a yields the result r, and the same function is applied to sequence b that starts with a, then the result will start with r. The monotonicity describes an important property of the execution of processes, viz. that a process cannot change the output it has already produced. The functions must also be Scott-continuous, which on the prefix order means that in addition to being monotonic, a function may not depend on whether an input sequence is finite or infinite. Kahn processes are by construction Scott-continuous functions.

A network of processes is described as an equation system where the data on the communication channels are variables and the processes are continuous functions over these variables. The semantics of the program is the smallest solution to the equation system with respect to the prefix order. This solution is unique and can be computed, or if infinite, arbitrarily well approximated, with fixed point iteration. A consequence of the solution being unique is that the execution order of the processes cannot affect the data they produce.

III. Cal

CAL [10] is a language for describing actors of dataflow with firing, originally developed as part of the Ptolemy project [11]. A variant of CAL has been standardized by MPEG in ISO/IEC 23001-4:2014 for describing video codecs.

An actor consists of *ports*, *state*, *actions* and additional constraints on when actions can be fired. Figure 2 shows an actor written in CAL. It has three actions, tagged with `start`, `add` and `done`. Two of the actions also have a guard, i.e. a boolean expression that must be true for the action to fire. The actions are also, in this example, constrained by an action schedule—a finite state machine that controls which actions can be fired.

CAL can express computation that is not possible using Kahn processes, namely actors that are not monotonic on the prefix order or not even deterministic. The `Merge` actor in figure 3 is an example of a non-deterministic actor. It takes a token either from port `X` or port `Y` and puts its value on `Z`.

```
actor Merge() X, Y ⟹ Z :
  action X:[v] ⟹ Z:[v] end
  action Y:[v] ⟹ Z:[v] end
end
```

Fig. 3. Non-deterministic merge actor

$$\begin{tikzcd}
\text{process} \arrow[r, "A"] & \text{flowchart} \arrow[r, "B"] \arrow[d, "C"] & \text{denotational semantics} \\
\text{actor} & \text{flowchart with actions} \arrow[l, "D"'] \arrow[r, "B'"] & \text{denotational semantics with firing}
\end{tikzcd}$$

Fig. 4. An overview of the transformations presented in this paper.

IV. Process Language

In this section we describe a small language for writing Kahn processes, which we in the next section translate to dataflow with firing. Figure 4 shows an overview of the transformations that are performed on the processes. We will refer to this figure for each transformation we introduce.

A. Language Grammar

The grammar of the language is shown in figure 5. Some parts of the grammar are not described in this paper, namely `expression`, `identifier` and `type`, and for those productions we use the corresponding productions from the CAL language report [10]. The process example in figure 1 is written in this language, and later in the paper, there are a few more examples.

A process begins with the keyword **`process`** followed by the name of the process. Then follows a list of formal parameters in parentheses, and the actual parameters for these are bound at compile time, when instantiating the dataflow graph. After the parameter list comes the input and output port declarations. These are also bound to the communication channels when instantiating the graph.

The body of the process starts with a list of variable declarations, followed by the process description. The process description is either repeated, indicated by the **`repeat`** keyword, or just executed once, in case the **`begin`** keyword is used. The statements in the process description is what the process executes at runtime.

There are five kinds of statements of which three of them are known from many other imperative programming languages: if statements, while loops and assignments. The read and write statements are central for the semantics of Kahn processes. The read statements look like this

```
Port ⟶ variable
```

and reads one token from `Port`, which must be an input port, and assigns the value to `variable`. The read is blocking, which means that the execution will not proceed unless there

```
entity = "process" identifier "(" parameters ")"
  ports "==>" ports ":"
  {declaration}
  process
  "end".

parameter = [type] identifier.
parameters = [parameter {"," parameter}].

port = [type] identifier.
ports = [port {"," port}].

declaration = [type] identifier
  [":=" expression] ";".

process = ("repeat" | "begin") {statement} "end".

statement = if | while | read | write | assignment.

if = "if" expression "then" {statement}
  ["else" {statement}] "end".

while = "while" expression "do" {statement} "end".

read = identifier "-->" identifier ";".

write = identifier "<--" expression ";".

assignment = identifier ":=" expression ";".
```

Fig. 5. Simplified grammar of a process language

is a token available. The write statements, however, are non-blocking and look like this

```
Port ⟵ expression
```

The expression is evaluated and the value is written to `Port`, which must be an output port.

B. Semantics

The semantics of this language is the semantics of Kahn processes, as described in [1], where the processes are viewed as functions on sequences. We use $[v_1, v_2, \ldots, v_n]$ to denote a sequence of n elements, and $[v_1, v_2, \ldots]$ to denote a sequence of infinite length. Concatenation, denoted $X.Y$, is the sequence that starts with X and continues with Y, or just X if X is infinite. We use $[]$ to denote an empty sequence.

The first step towards describing a process as a function on sequences is to describe it as a flowchart with one node in the flowchart per statement. This step is labelled A in the overview in figure 4.

The read and write statements are represented by input/output nodes (parallelograms). Assignments are represented by process nodes (rectangles). The branching statements **`if`** and **`while`** are represented by groups of nodes: one decision node (diamond) for the condition test, one subchart for each alternative execution path, and a connection node (small circle) where the control flow converges. Entry points and exit points are represented by terminal nodes (rounded rectangles), labeled start and stop, respectively.

The flowchart is then translated to a function on sequences. This is transformation B in the overview in figure 4. The denotational semantics of Kahn processes is defined on this form, and Kahn refers to the methods of McCarthy [12] on how to do the translation.

Each node in the flowchart is translated to a function that refers to its successor nodes for the continued execution. The functions are parameterized by the state variables and input streams. For simplicity, we only consider one state variable v, one input sequence X and one output sequence, but later we generalize this to tuples. A start node *start* initializes the state of the process,

$$\mathit{start}(v, X) = \mathit{next}(v', X) \tag{1}$$

where v' is the initial value of v and *next* represents the successor node in the flowchart. A stop node *stop* represents the end of the execution and is therefore the empty sequence.

$$\mathit{stop}(v, X) = []$$

A connection node *conn* is defined as its successor *next* and can be omitted. An assignment node is defined as

$$\mathit{assign}(v, X) = \mathit{next}(f(v), X)$$

where f computes the new value of v and *next* is the successor node. A conditional node, *cond* has two possible successor nodes, *true* and *false*, and is defined as

$$\mathit{cond}(v, X) = \begin{cases} \mathit{true}(v, X), & \text{if } p(v) \\ \mathit{false}(v, X), & \text{otherwise} \end{cases}$$

where p is the predicate on the state variables that is the condition of the node. A write node *write* that writes a value $f(v)$ derived from the state to the output port is defined as follows.

$$\mathit{write}(v, X) = [f(v)].\mathit{next}(v, X) \tag{2}$$

Finally, a read node that reads a token and assigns it to v is defined as

$$\mathit{read}(v, X) = \begin{cases} \mathit{next}(h, T) & \text{if } X = [h].T \\ [] & \text{if } X = [] \end{cases} \tag{3}$$

where the execution stops if there is no token available. The description of a process is completed by a function that hides the state variables and is defined as the start function with any values of the variables.

$$\mathit{process}(X) = \mathit{start}(\perp, X)$$

If a process function is constructed using only the functions described above, the process will be monotonic on the prefix relation, because extension of an input to the function can only affect extensions of its output. It will also be continuous, because the function cannot depend on the finiteness of the sequences. The function therefore represents a Kahn process.

To generalize this to handle more state variables and input ports and output ports, the variables can be represented by a tuple, and the input sequences by a tuple of sequences and the output by a tuple of sequences as well. We extend the prefix order to tuples of sequences with $(X_1, \ldots, X_n) \sqsubseteq (Y_1, \ldots, Y_n) \iff X_1 \sqsubseteq Y_1 \wedge \ldots \wedge X_n \sqsubseteq Y_n$ and define the concatenation operation $(X_1, \ldots, X_n).(Y_1, \ldots, Y_n)$ as elementwise concatenation $(X_1.Y_1, \ldots, X_n.Y_n)$. The *assign* function will only change one of the elements of the variable tuple, *read* will only read from one of the streams to one of the variables, and *write* will only add to one of the elements of the output sequence tuple.

As an example of this translation, the `Delay` in figure 6 is first transformed to the flowchart in figure 7. The nodes of this flowchart are numbered 1 to 3, which corresponds to the functions Delay_1 to Delay_3. These functions are defined as described above.

$$\mathit{Delay}_1(v, X) = \mathit{Delay}_2(0, X) \qquad \text{by (1)}$$
$$\mathit{Delay}_2(v, X) = [v].\mathit{Delay}_3(v, X) \qquad \text{by (2)}$$
$$\mathit{Delay}_3(v, X) = \begin{cases} \mathit{Delay}_2(h, T) & \text{if } X = [h].T \\ [] & \text{if } X = [] \end{cases} \qquad \text{by (3)}$$

The process is defined as follows.

$$\mathit{Delay}(X) = \mathit{Delay}_1(\perp, X)$$

We can now simplify function Delay_3 by substituting Delay_2

$$\mathit{Delay}_3(v, X) = \begin{cases} [h].\mathit{Delay}_3(h, T) & \text{if } X = [h].T \\ [] & \text{if } X = [] \end{cases}$$

Now we can see that Delay_3 is equivalent to its X argument and further simplify it to

$$\mathit{Delay}_3(v, X) = X$$

Similarly we can simplify *Delay* by substituting Delay_1 and then Delay_2 and then finally Delay_3 and get the following.

$$\mathit{Delay}(X) = [0].X$$

This definition is a very compact description of the process that captures the essence of the unit delay.

```
process Delay () X ⟹ Y :
  v := 0;
  repeat
    Y ⟵ v;
    X ⟶ v;
  end
end
```

Fig. 6. A process describing the unit delay.

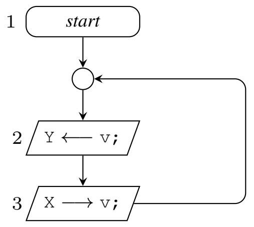

Fig. 7. A flowchart of the `Delay` process in figure 6 with numbered nodes.

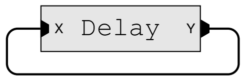

Fig. 8. A cyclic dataflow program with the `Delay` process.

Figure 8 shows a small cyclic network with one `Delay` process. The corresponding equation system in the semantics of Kahn processes is

$$\begin{cases} Y = \mathit{Delay}(X) \\ X = Y. \end{cases} \tag{4}$$

In general, there are several solutions to the equation systems of Kahn process networks, but the smallest solution with respect to the prefix order is what defines the semantics. In this case, the smallest solution to the equation system in (4) has an output sequence that is an infinite sequence of zeros: $Y = [0, 0, 0, \ldots]$.

V. Translation to dataflow with firing

The translation from process to dataflow with firing is done in three steps. (The corresponding labels from figure 4 are shown in parentheses.)

1) Construct a flowchart of the process. (A)
2) Group nodes in the flowchart into actions. (C)
3) Create a dataflow actor from the actions. (D)

The first step makes it easier to reason about the statements in the code. The second step cannot be done arbitrarily without the risk of changing the semantics of the program. We therefore show how this step can be performed without changing what the process computes by making sure that the result of B and B' in figure 4 are the same. In the third step, we take a process with actions and create a CAL actor with the same control flow.

A. Grouping nodes into actions

A dataflow actor is executed in atomic steps, called action firings. An action can only be fired if its conditions are fulfilled. We call these *firing conditions*. The tokens that an action reads, for example, must be present in order to fire. It also means that the number of tokens that an action requires must be known before firing that action. Typically, but not necessarily, it is even known at compile-time.

An example of a possible action is a chain of read, write and assignment nodes in the flowchart. Because a chain only has one possible control flow, where each statement is executed exactly once, the number of tokens that will be consumed and produced by executing a chain is known at compile-time. However, an action grouping is not correct just because it contains valid actions. It must also represent the same sequence-function.

To determine if a process with its statements grouped to actions is equivalent to the original process, we add action firing to the functions on sequences and check if the function is still the same. Referring to figure 4, we check that the results of B and B' are the same.

If an action a is a sequence of statements that start with s and reads n tokens from X, then the atomicity is modeled as

$$a(v, X) = \begin{cases} s(v, X) & \text{if } X = [h_1, \ldots, h_n].T \\ [\,] & \text{otherwise.} \end{cases} \tag{5}$$

The execution "continues" only when the input sequence X is long enough to execute the whole sequence of statements. The definition in (5) can be generalized to handle more than one input sequence by adding more firing conditions.

As an example, we use the `Delay` process in figure 6 and its corresponding flowchart in figure 7. Let the write statement of node 2 together with the read statement of node 3 be an action. The requirement for this action to fire is that X has at least one element, because of the read statement. The effect of making an action of statement 2 and 3 is that the write will not be executed unless there is a token available for the read. To show that this translation is incorrect, we construct the function that corresponds to this translated process.

$$\begin{aligned}
\mathit{Delay}'(X) &= \mathit{Delay}'_1(\bot, X) \\
\mathit{Delay}'_1(v, X) &= \mathit{Delay}'_a(0, X) && \text{by (1)} \\
\mathit{Delay}'_a(v, X) &= \begin{cases} \mathit{Delay}'_2(v, X) & \text{if } X = [h].T \\ [\,] & \text{if } X = [\,] \end{cases} && \text{by (5)} \\
\mathit{Delay}'_2(v, X) &= [v].\mathit{Delay}'_3(v, X) && \text{by (2)} \\
\mathit{Delay}'_3(v, X) &= \begin{cases} \mathit{Delay}'_a(h, T) & \text{if } X = [h].T \\ [\,] & \text{if } X = [\,] \end{cases} && \text{by (3)}
\end{aligned}$$

This set of functions differs from the Delay functions in the following two ways. Delay' contains a function Delay'_a that describes the atomicity of Delay'_2 and Delay'_3, and all references to Delay_2 are instead references to Delay'_a.

A translation is correct only if its function is equivalent to the original function. If we apply Delay and Delay' to the empty sequence

$$\begin{aligned}
\mathit{Delay}([\,]) &= [0] \\
\mathit{Delay}'([\,]) &= \mathit{Delay}'_1(\bot, [\,]) = \mathit{Delay}'_a(0, [\,]) = [\,]
\end{aligned}$$

we see that the translation is incorrect.

B. Action grouping schemes

We will present a series of action grouping schemes that yield correct translations from processes to dataflow with firing. We only consider grouping schemes that group chains of statement nodes, i.e. sequences of read, write and assignment statements. The condition nodes are translated to their own actions without any firing conditions that only designate its successor action in the execution.

There is a trivial action grouping that is always correct, that is the grouping where each statement becomes its own action. In this grouping, only the actions with a read statement will have a firing condition. This condition is also the same as the condition of the read statement itself. Let a be the action with one read statement r,

```
X ⟶ v;
```

whose successor is n.

$$a(v,X) = \begin{cases} r(v,X) & \text{if } X = [h].T \\ [\,] & \text{if } X = [\,] \end{cases}$$

$$r(v,X) = \begin{cases} n(h,T) & \text{if } X = [h].T \\ [\,] & \text{if } X = [\,] \end{cases}$$

We can see that $a = r$. For an action a with a write or assignment statement s, the action is equal to the statement, because the action has no firing condition.

$$a(v,X) = \begin{cases} s(v,X) & \text{always} \\ [\,] & \text{never} \end{cases}$$

This grouping results in very fine grained actions, but dataflow actors are usually not written in this way, firstly because it is very tedious to write that many actions, secondly because it is usually not necessary, and thirdly because it is usually more efficient to write fewer and larger actions. The actions need to be scheduled, either at compile time or at runtime, and the more actions to schedule, the more time it tends to take. For these reasons, we study some grouping schemes with larger actions as well.

1) Regular expressions on sequences of statements.: To describe different grouping schemes, we use regular expressions over sequences of statements. Let r, w and a denote the set of read, write or assignment statement, respectively. Alternation is denoted $x|y$ which is the union of all sequences in x and y. Concatenation xy is the set of sequences that are concatenations of a sequence in x with a sequence in y. The Kleene star x^* represents concatenation of any number of sequences in x, including zero sequences. As an example, the expression $(r|a)^*$ denotes any sequence of read and assignment statements.

2) Actions with only writes and assignments.: If sequences of write and assignment statements, $(w|a)^*$, are grouped to actions, the resulting function of the action will be identical to the function of the statements. Write and assignment statements do not read any tokens, and the action will therefore have no conditions. Let s be the first statement in a sequence in $(w|a)^*$, followed by the rest of the process, and a be an action with these statements.

$$a(v,X) = \begin{cases} s(v,X) & \text{always} \\ [\,] & \text{never} \end{cases}$$

Action a is trivially equal to statements s.

3) Actions with only reads and assignments.: If sequences of read and assignment statements, $(r|a)^*$, are grouped to actions, the resulting function will have conditions on the lengths of the sequences corresponding to the number of read statements for each stream. Assume the read statements of an action a reads n tokens from X, the action will then be defined as

$$a(v,X) = \begin{cases} s(v,X) & \text{if } X = [h_1,\dots,h_n].T \\ [\,] & \text{otherwise,} \end{cases}$$

where s represents the execution starting at the first statement in the action. Since the statements of the action doesn't produce any output, we know that when the sequence is shorter than what all statements read, the output is empty.

$$\forall k < n.\, s(v,[x_1,\dots,x_k]) = [\,]$$

That is also true for the action.

$$\forall k < n.\, a(v,[x_1,\dots,x_k]) = [\,]$$

When the sequences, however, are long enough for all the read statements of the action, we get the following.

$$\forall k \geq n.\, a(v,[x_1,\dots,x_k]) = s(v,[x_1,\dots,x_k])$$

Both when the inputs are sufficiently long and when they are not, the action represents the same function as its statements.

4) Actions with reads and assignments and then writes and assignments.: The two regular expressions above can be combined to $(r|a)^*(w|a)^*$, accepting first reads and assignments and then writes and assignments. To study what affect this action has on the function, we first study the case with two consecutive actions a_1 with statements from $(r|a)^*$ and a_2 with statements from $(w|a)^*$. As we have seen earlier, action a_1 requires all tokens that it reads to be present before proceeding, and action a_2 does not require anything to continue. An action a that contains the statements of a_1 followed by a_2, will have the same conditions as a_1, because a_2 does not have any conditions. The function that represents a will therefore be identical to the function that represents a_1 followed by a_2. To illustrate this, assume the simple case with one read statement r followed by one write statement w followed by another statement n. First, we make actions a_r of r and a_w of w. The functions for this now looks like the following.

$$a_r(v,X) = \begin{cases} r(v,X) & \text{if } X = [h].T \\ [\,] & \text{if } X = [\,] \end{cases}$$

$$r(v,X) = \begin{cases} a_w(h,T) & \text{if } X = [h].T \\ [\,] & \text{if } X = [\,] \end{cases}$$

$$a_w(v,X) = w(v,X)$$

$$w(v,X) = [v].s(v,X)$$

The important part is that a_w will be the same for all groupings in $(w|a)^*$, making it possible to use w directly wherever a_w is used and eliminate a_w.

$$a_r(v,X) = \begin{cases} r(v,X) & \text{if } X = [h].T \\ [\,] & \text{if } X = [\,] \end{cases}$$

$$r(v,X) = \begin{cases} w(h,T) & \text{if } X = [h].T \\ [\,] & \text{if } X = [\,] \end{cases}$$

$$w(v,X) = [v].s(v,X)$$

If instead we make one action a of both r and w we get the same functions.

$$a(v,X) = \begin{cases} r(v,X) & \text{if } X = [h].T \\ [\,] & \text{if } X = [\,] \end{cases}$$

$$r(v,X) = \begin{cases} w(h,T) & \text{if } X = [h].T \\ [\,] & \text{if } X = [\,] \end{cases}$$

$$w(v,X) = [v].s(v,X)$$

Grouping statements by $(r|a)^*(w|a)^*$ to actions preserves the semantics of the process.

C. Translation to dataflow actor

From the flowcharts with actions, the translation to dataflow with firing (transformation D in figure 4) is quite straightforward. We use CAL as the target language for this last step, and we explain the translation with the SumN process in figure 1 and the actions of figure 9

When the actions of a process are identified, they are translated to CAL actions that perform the same task, and each action gets a unique action tag. We call these actions *statement actions*. From the grouping in figure 9, the following actions are constructed.

```
a:
action N:[temp] ⟹
do
  n := temp;
  sum := 0;
end

b:
action X:[temp] ⟹
do
  x := temp;
  sum := sum + x;
  n := n - 1;
end

c:
action ⟹ Sum:[sum] end
```

From the decision node, two actions are constructed, one that can be fired when the condition is true, and one when the condition is false. We call these pairs of *condition actions*. Note that these actions neither consume nor produce any tokens.

```
d_true:
action ⟹
guard
  n > 0
end

d_false:
action ⟹
guard
  not (n > 0)
end
```

Finally, an action schedule is created to ensure that the actor has the same control flow as the original process. The schedule will have one state for each statement action, and one state for each pair of condition actions. The transitions are then constructed to reflect the control flow of the process.

```
schedule A :
  A (a) ⟶ D;
  D (d_true) ⟶ B;
  D (d_false) ⟶ C;
  B (b) ⟶ D;
  C (c) ⟶ A;
end
```

The execution starts in state A with action a that reads how many numbers should be summed, and continues to state D. In D, there are two actions, one that can be executed when $n > 0$ and another action otherwise. Depending on which of the two is fired, the execution proceeds to either B or C. State B represents the body of the loop and when action b is fired the execution proceeds with the loop condition in state D again. After the loop, in state C, action c is executed an the output is produced. The execution then continues in state A to start a new summation.

The start node of the flowchart corresponds to the variable initialization of the process, and this is just copied to the resulting actor. This example does not have any stop node, but stop nodes are represented by states in the schedule without any transitions that leaves them.

The end result is not as polished as the handwritten CAL version in figure 2, but it is not far from it.

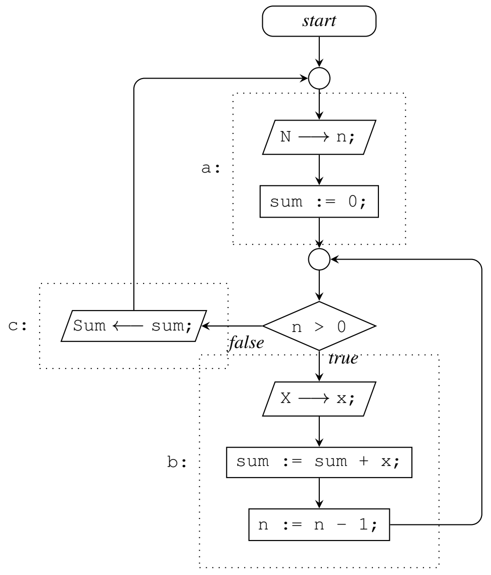

Fig. 9. Flowchart of the SumN process in figure 1 with an example grouping.

VI. Discussion

A. Implementation efficiency

Even though this paper is not about efficient implementations of dataflow with firing, its existence is highly relevant for this work. Some restricted classes of dataflow with firing, such as synchronous dataflow and cyclo-static dataflow are well known for their efficient implementation techniques, but even dynamic dataflow can be efficienly implemented. To demonstrate the efficiency compared to a traditional process implementation, we implemented the proposed language and

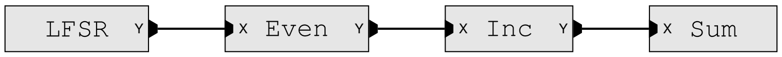

Fig. 10. A program with four processes. `LFSR` is a linear feedback shift register, producing the $n = 10^7$ first numbers of a maximum length sequence of 16 bits, followed by a 0. `Even` filters out all odd values, forwarding only the even numbers to `Inc`, that increments the numbers by 1. `Sum` computes the sum of all numbers until it gets a 1 (the incremented 0 at the end of the stream) and prints the sum.

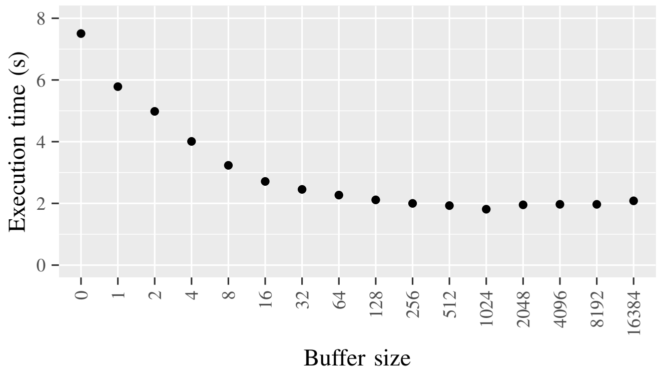

Fig. 11. Execution time of the Go program with different buffer sizes, where 0 indicates rendezvous communication.

TABLE I
EXECUTION TIME OF THE PROGRAM IN FIGURE 10

	Mean (ms)	Standard deviation (ms)
Go (gc)	1811.4	13.6
Kahn processes (Tÿcho)	203.0	3.7
Kahn processes (Tÿcho)[a]	158.3	2.8
Kahn processes (Tÿcho)[b]	64.9	2.5
C (Clang)[c]	64.6	2.8

[a] compiled with actor merging
[b] compiled with actor merging and buffer to variable conversion
[c] a sequential program that computes the same result

the translation to CAL in the Tÿcho dataflow compiler and designed a small program with four processes, depicted in figure 10. We made two process implementations; one in our proposed language that we compiled using the Tÿcho dataflow compiler, and one with goroutines in Go that we compiled using gc[1]. To get an indication of how much overhead the parallel descriptions induces in terms of scheduling and bookkeeping, we also made a sequential implementation in C that performs the same computation using a single loop.

In the Go implementation, the size of the channels that connects the goroutines affects the performance significantly. Figure 11 shows the execution time for buffer sizes ranging from 0 to 2048, where 0 means rendezvous communication. Using large buffers in the program implemented with Tÿcho did not result in any significant performance difference.

The Tÿcho dataflow compiler can merge actors using actor machine composition, as described in [7], similar to what is done with RVC-CAL actors using the Open RVC-CAL Compiler (Orcc) in [8]. The resulting merged actor does not need to check that state of the channels that are connected to itself. If these self-loop channels are of size 1, they can be replaced by variables, which is also done by Tÿcho. Table I shows the execution time of five different implementations:

1) the Go implementation with channel sizes set to 1024,
2) the Kahn process implementation compiled with Tÿcho,
3) the Kahn process implementation compiled with Tÿcho with actor merging,
4) the Kahn process implementation compiled with Tÿcho with actor merging and with buffers converted to variables, and
5) the sequential C program.

The Kahn process implementations that are transformed to dataflow with firing are clearly faster than the goroutine implementation, even without actor merging. With actor merging and the buffer to variable transformation, the programs is as fast as the sequential C program.

All measurements were performed on a computer with a quad-core 2 Ghz Intel Core i7 processor running OS X, and the sequential C program and the C programs produced by Tÿcho were compiled with Clang using the flag `-O2`.

B. Program simplicity

We believe that many dataflow actors could be much simpler described as processes, most importantly because the control flow of a process better follows the text flow of the source program than it would do in a dataflow-with-firing style. To get an indication on what kinds of actors would benefit from being expressed as processes, we have identified some patterns commonly used in CAL actors by studying the example applications from *orc-apps*[2]—a collection of CAL applications. The patterns we identified are the following:

1) Actors with only one action.
2) Fixed sequences of actions in the sense that action B always follows action A.
3) Iterative token production or consumption with an unknown number of tokens.
4) Actors that select different actions depending on the value of a particular token or state variable.

In our opinion, pattern 1 is often clearly expressed in CAL and we see no point in rewriting such actors in a process style. However, expressing pattern 2, 3 and 4 is, in our opinion, more complex in CAL than it is in our process language. Fixed sequences of actions (pattern 2) involve at least two actions and an action schedule. When trying to follow the control flow of such an actor, the reader must consult the action schedule to see in which order the actions are fired. When expressed as a process, this pattern is simply a sequence of statements.

[1]The other official Go compiler—gccgo—produced slower programs for this example.

[2]https://github.com/orcc/orc-apps

TABLE II
NUMBER OF TOKENS IN THE SOURCE CODE OF THE ACTORS AND THEIR RESPECTIVE PROCESSES.

	Actor	Proc.	Ratio	Patterns
SyntaxParser[a]	682	412	60%	2, 3, 4
Mgnt_DCSplit[b]	97	107	110%	1
Algo_Byte2bit[b]	122	81	66%	2, 3
Algo_SelectMB_4[c]	491	200	41%	2, 3, 4
Algo_PictureRecon…Sat…[b]	355	255	72%	2, 4
KeySchedule[e]	757	629	83%	2

[a] from JPEG decoder
[b] from MPEG-4 Part 2 decoder
[c] from MPEG-4 AVC decoder
[d] from AES cipher

Pattern 3 and 4 can in our process language be expressed by (respectively) a **while**-loop and **if**-statement, but encoding the same pattern in a CAL actor requires at least two actions whose firing conditions only differ in a guard expression, representing the branching condition.

A review of existing CAL applications and the feasibility of translating them to processes is out of the scope for this paper. We have, however, collected a few actors that use the patterns described above and translated them to our process language. We have not measured how readable they are, or how well the program text follows the control flow, but since the simplified control flow also results in smaller programs, we have measured the size of the programs in number of source code tokens. As the result in table II shows, the actor with only one action (pattern 1) uses more tokens in the process version than the actor version, but the other patterns are expressed using fewer tokens in their process versions.

C. Language Design

The first objective of the language design is to enable programmers to write processes and have them executed as actors. The second objective is to make it easy to use processes in CAL programs. We have made some design choices motivated by these objectives.

The look and feel of the language resembles CAL to a large extent. The reason for this decision is to make it easier for CAL programmers to start using the language. Not only does it look like CAL, the types, statements and expressions are borrowed from CAL as well. By representing and computing values in the same way, the interaction between the two languages becomes straightforward. In the larger version of the language that is implemented in the Tÿcho dataflow compiler, the process description is actually implemented as a new language construct for CAL actors, rather than a separate language of its own.

D. More general groupings

We saw in the *Delay′* example that actions that start with writes and continue with reads are not equivalent to just the statements themselves. There are, however, other correct grouping schemes that recognize larger actions. The schemes we have looked at all consist of sequences of statement nodes, but there are examples where loops and conditionals can be part of actions as well. If, for example, the two execution paths of an **if** statement have the same read and write pattern, they could be considered for inclusion in the same action. The same is true for loops where no communication happens. The grouping schemes that we have shown in this paper are therefore not the most coarse-grained.

VII. RELATED WORK

A. Language

The process language we introduced makes it possible to write dataflow programs as combinations of processes and dataflow with firing, but it is not the first example of such combination. StreamIt [13] is a language for Synchronous dataflow—a flavor of dataflow with firing where the number of tokens consumed and produced by an actor is the same on every invocation. In version 2.1 of StreamIt the restriction of a fixed input and output rate was lifted, effectively making StreamIt a language for expressing both synchronous dataflow and Kahn processes. The implementation in [5] also treats them separately, heavily optimizing the parts with fixed token rates and dynamically scheduling all interaction that deals with dynamic token rates. Our approach is instead to unify the two by translating the process to dataflow with firing. In StreamIt, however, the filters with dynamic token rates can not in general be translated to their current model for dataflow with firing, synchronous dataflow, because that model is not expressive enough.

B. Translation

In [14], Falk et al. presents a translation from Kahn processes expressed in SystemC to dataflow with firing by constructing a control flow graph in which all read statements precede the write statements in the basic blocks. This is equivalent to the most general action grouping scheme presented in this paper. The novelty of our work is the connection to the semantics of Kahn processes to show that the action grouping is correct.

One aspect of this work is that it enables programmers to write programs in an imperative style and have it executed with a different execution model that we believe is better in many ways. Related to this, Capriccio [15] is a threading package for C that enables programmers to write threaded code with blocking operations, have it executed using cooperatively scheduled coroutines and event-based operations instead of blocking ones. Similarly, Tame [16] and the work of Adya et al. [17] both try to unify the models of threaded with event-driven programs, partly using program transformations not unlike those we do.

Even though there are similarities between these systems and ours, we believe that the challenges differ very much. One of their main challenge is memory management for automatic variables. That is not an issue for our language, because the processes are essentially stack-free at the points where they need to be. Our main challenge, on the other hand, is to make sure that the exact semantics of Kahn processes is retained in the translation.

In [18], Lee shows a translation from dataflow with firing to Kahn processes, which is the reverse of our translation, and gives conditions on when this translation is possible.

C. Process model

In this work, we use Kahn process networks as our process model. There are other well-known process models, such as Communicating Sequential Processes [19] or pi calculus [20]. We chose Kahn process networks over these calculi because the communication model of Kahn process networks closely resembles the one of dataflow with firing. Because of this similarity, several aspects of combining Kahn process networks and dataflow with firing have already been studied. In [18], Lee shows that a certain class of dataflow with firing is determinate by transformation to Kahn process networks. In [21], Kienhuis and Deprettere introduce a model for dataflow with firing that is able to describe Kahn process networks.

When proving the equivalence of the programs before and after action grouping, we extended the Kahn process model with a notion of actions. There is a related model, called Stream Based Functions [22] that instead extends a model for dataflow with firing with the expressiveness of Kahn processes. This model is used in Compaan [23] to transform nested loop programs to Kahn process networks.

VIII. Conclusions and Future Work

We have introduced a Kahn process language and a family of translations from this language to dataflow with firing and CAL. It enables programmers to write their dataflow programs as processes and still use the efficient implementation techniques and analysis tools that available for CAL. For software implementations, if the number of processes exceeds the number of processors, which is the most common case, their corresponding actors can be merged to reduce communication overhead using actor machine composition [7] or by merging CAL actors [8]. For programmable hardware, Xronos have been shown to synthesize hardware directly from the RVC-CAL reference implementation of an MPEG-4 decoder [24]. On the analysis side, the profiling infrastructure of TURNUS uses execution traces of CAL programs to explore the design space of their implementations [9].

We have also elaborated on the translation from a Kahn process language to its denotational semantics, introducing a way of reasoning about process languages, building upon the works of Kahn [1] and in turn McCarthy [12]. This way of reasoning enabled us to introduce the concept of an action to the semantics of Kahn processes.

To conclude the paper, we have shown a way of giving programmers the simplicity of writing processes and at the same time treat them as CAL actors with all of its benefits. As future work, on the language side, we would like to investigate ways of letting programmers affect the action grouping. We would also like to integrate more features of CAL into the process language and vice versa. On the translation side, we would like to study action grouping schemes that can identify even larger actions.

References

[1] G. Kahn, "The semantics of a simple language for parallel programming," in *In Information Processing'74: Proceedings of the IFIP Congress*, vol. 74, 1974, pp. 471–475.

[2] G. Kahn and D. MacQueen, "Coroutines and networks of parallel processes," 1976.

[3] E. Lee, D. G. Messerschmitt *et al.*, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[4] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cycle-static dataflow," *Signal Processing, IEEE Transactions on*, vol. 44, no. 2, pp. 397–408, 1996.

[5] R. Soulé, M. I. Gordon, S. Amarasinghe, R. Grimm, and M. Hirzel, "Dynamic expressivity with static optimization for streaming languages," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 2013, pp. 159–170.

[6] R. Gu, J. W. Janneck, M. Raulet, and S. S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," *Journal of Signal Processing Systems*, vol. 63, no. 1, pp. 129–142, 2011.

[7] G. Cedersjö and J. W. Janneck, "Software code generation for dynamic dataflow programs," in *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*. ACM, 2014, pp. 31–39.

[8] J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silven, "Actor merging for dataflow process networks," *Signal Processing, IEEE Transactions on*, vol. 63, no. 10, pp. 2496–2508, 2015.

[9] S. Casale-Brunet, A. Elguindy, E. Bezati, R. Thavot, G. Roquier, M. Mattavelli, and J. W. Janneck, "Methods to explore design space for mpeg rmc codec specifications," *Signal Processing: Image Communication*, vol. 28, no. 10, pp. 1278–1294, 2013.

[10] J. Eker and J. W. Janneck, "CAL language report: Specification of the CAL actor language," December 2003.

[11] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," 1994.

[12] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[13] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *International Conference on Compiler Construction*, Grenoble, France, Apr 2002.

[14] J. Falk, C. Zebelein, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya, "Analysis of systemc actor networks for efficient synthesis," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 10, no. 2, p. 18, 2010.

[15] R. Von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: scalable threads for internet services," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 268–281.

[16] M. N. Krohn, E. Kohler, and M. F. Kaashoek, "Events can make sense." in *USENIX Annual Technical Conference*, 2007, pp. 87–100.

[17] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative task management without manual stack management." in *USENIX Annual Technical Conference, General Track*, 2002, pp. 289–302.

[18] E. A. Lee, *A denotational semantics for dataflow with firing*. Electronics Research Laboratory, College of Engineering, University of California, 1997.

[19] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[20] R. Milner, *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[21] B. Kienhuis and E. F. Deprettere, "Modeling stream-based applications using the sbf model of computation," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 34, no. 3, pp. 291–300, 2003.

[22] A. C. J. Kienhuis, *Design space exploration of stream-based dataflow architectures*. TU Delft, Delft University of Technology, 1999.

[23] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: Deriving process networks from matlab for embedded signal processing architectures," in *Proceedings of the eighth international workshop on Hardware/software codesign*. ACM, 2000, pp. 13–17.

[24] E. Bezati, S. C. Brunet, M. Mattavelli, and J. W. Janneck, "Synthesis and optimization of high-level stream programs," in *Electronic System Level Synthesis Conference (ESLsyn), 2013*. Ieee, 2013, pp. 1–6.