Executing Dataflow Actors as Kahn Processes

Andreas Tretter* Jani Boutellier‡ James Guthrie* Lars Schor* Lothar Thiele*

*Computer Engineering and Networks Laboratory
ETH Zurich, Zurich, Switzerland
firstname.lastname@tik.ee.ethz.ch

‡Department of Computer Science and Engineering
University of Oulu, Oulu, Finland
jani.boutellier@ee.oulu.fi

ABSTRACT

Programming models which specify an application as a network of independent computational elements have emerged as a promising paradigm for programming streaming applications. The antagonism between expressivity and analysability has led to a number of different such programming models, which provide different degrees of freedom to the programmer. One example are Kahn process networks (KPNs), which, due to certain restrictions in communication, can guarantee determinacy (their results are independent of timing by construction). On the other hand, certain dataflow models, such as the CAL Actor Language, allow non-determinacy and thus higher expressivity, however at the price of static analysability and thus a potentially less efficient implementation. In many cases, however, non-determinacy is not required (or even not desired), and relying on KPN for the implementation seems advantageous.

In this paper, we propose an algorithm for classifying dataflow actors (i.e. computational elements) as KPN compatible or potentially not. For KPN compatible dataflow actors, we propose an automatic KPN translation method based on this algorithm. In experiments, we show that more than 75% of all mature actors of a standard multimedia benchmark suite can be classified as KPN compatible and that their execution time can be reduced by up to 1.97x using our proposed translation technique. Finally, in a manual classification effort, we validate these results and list different classes of KPN incompatibility.

Keywords

Dataflow programming, Kahn process networks, classification

1. INTRODUCTION

The critical importance of multi-processor technology in future data-processing systems has led to parallel programming models being extensively studied in the last two decades. One promising paradigm that has emerged for programming streaming and multimedia applications is to specify the structure of applications as directed graphs, the functionality of which is split into a set of independent computational elements that can only communicate through point-to-point first in, first out (FIFO) channels. This helps to avoid the need for additional synchronisation as well as many pitfalls of parallel execution such as data races. At the same time, this paradigm explicitly exposes concurrency in the application, thus considerably simplifying execution on a parallel system.

Two popular classes of programming models following this paradigm are dataflow models [14] and Kahn process networks (KPNs) [12]. A major difference between them lies in the way they describe the computational elements, *Kahn processes* and *dataflow actors*. While for Kahn processes, the only way of obtaining input is to first choose a channel for reading from and then to wait until data arrive, advanced dataflow actors may base their reading behaviour on the availability of input data on the channels and even on this data itself (*peeking*). As an implementational example for this kind of dataflow networks, we focus in this paper on the CAL actor language [9], of which a subset named RVC-CAL [16] has been standardized by ISO/IEC 23001-4:2009 MPEG to specify multimedia applications. Clearly, these possibilities allow for a higher expressiveness and flexibility. On the other hand, if an actor's behaviour depends on data availability, it may also depend on timing. Consequently, an actor classification into either static, dynamic or time-dependent has been proposed [20, 22].

In contrast, KPNs are always determinate, i.e. the sequences of tokens on the channels do not depend on timing. KPNs bring the benefit of a more efficient low-level implementation, as a Kahn process is at any moment either computing or waiting for input from a specific channel. Its ready state can thus easily be determined, whereas for a CAL actor, multiple rules and criteria have to be evaluated first to achieve the same goal. This is one of the reasons why the KPN model has been widely used in high-level synthesis frameworks for parallel systems as, for instance, MAPS [7] or DAL [18]. Since even in non-determinate dataflow specifications, many actors are in fact determinate, it appears favourable to analyse these actors for KPN compatibility in order to exploit the related optimisation potential.

In this paper, we present a formal method for translating KPN compatible dataflow actors to Kahn processes. To this end, we first show that the aforementioned classification into static, dynamic, and time-dependent actors is inadequate for evaluating KPN compatibility. Afterwards, we propose an algorithm to classify a dataflow actor as KPN compatible or potentially KPN incompatible (it tries to identify sufficient conditions for KPN compatibility in a high number of real cases, the problem being undecidable in general). Based on this algorithm, we then propose a method to translate a KPN compatible dataflow actor to a Kahn process. Finally, we implement the proposed method in the RVC-CAL framework [21] and show that more than 75% of all mature actors of the RVC-CAL application suite [1] can be proven to be KPN compatible by our algorithm and that their performance can be improved by up to 1.97x when executing them as Kahn processes instead of dataflow actors. In a manual classification effort, we analyse the KPN compatibility of all these actors and list the different patterns and constellations leading to KPN incompatibility or non-recognition of KPN compatibility.

The remainder of the paper is organized as follows. In Section 2, an overview on the considered programming models is given. In Section 3, we describe the proposed translation technique. Experimental results are presented in Section 4. Finally, we review related work in Section 5.

978-1-4673-8079-9/15/$31.00

2. BACKGROUND

In this section, the dataflow and the Kahn process network (KPN) programming models are introduced in detail. In both of them, applications can be described as directed multi-graphs in which the nodes are computational elements and the edges, called *channels*, are unbounded FIFO buffers transporting so-called *tokens* from the source node to the destination node of the channel. The computational elements are independent from each other and can only communicate via channels. In KPNs these computational elements are called *processes* and in dataflow models they are called *actors*. The difference between dataflow graphs and KPNs is given by the different specification of actors and processes, which we will concentrate on in the rest of this work.

In the following, we will give a definition of a process according to Kahn and of an actor in dataflow models. Based on that, the considered problem of translating a dataflow actor into a Kahn process will be defined.

2.1 Kahn Processes

The KPN programming model [12] has a very generic definition of processes, imposing only the two restrictions: Firstly, each channel read access has to be *blocking* and *destructive*. *Blocking* means that when a process tries to read from an empty channel, it will wait (possibly for infinite time) until a token arrives on the channel. *Destructive* means that upon reading a token, the token is also removed from the channel. Secondly, the elementary operations performed in a process must not yield unpredictable results (e.g. access to timers or hardware random generators).

A Kahn process can be defined as follows:

Definition 1 A **Kahn process** π is a stateful, sequential program which performs calculations, write accesses to the outgoing channels and blocking, destructive read accesses to the incoming channels in any arbitrary sequence.

Due to this well-defined communication interface, Kahn processes are *determinate* in the sense that the sequences of tokens on their channels are independent of timing [12, 15, 14]. Determinacy can be formalised using the following two definitions:

Definition 2 Let γ be a channel. For any execution of the process network, the **history** of γ is the (possibly infinite) sequence of all tokens that traverse the channel, in the order in which they are written.

Correspondingly, the *input history* of a process is the combination of the histories of all its incoming channels and its *output history* that of its outgoing channels.

Definition 3 A process or an actor is **determinate** iff for any given input history, it will always produce the same output history.

2.2 Dataflow Actors

The behaviour of dataflow actors as defined in [14] is a repetition of so-called *firings*. In each firing, the actor will destructively read a specific amount of tokens from the incoming channels, perform computations and write a specific amount of tokens to the outgoing channels. An actor can only fire if all the input tokens are available on the channels.

The simplest form of dataflow is *synchronous dataflow* [13], where actors are stateless and each firing reads and writes a fixed number of tokens. Several extensions exist; in this work, we will concentrate on a very generic extension in which an actor may have a state and a set of different *actions* it can perform as a firing.

Algorithm 1: Illustration of the behaviour of a dataflow actor α.

```
while TRUE do
    A*_α ← {a ∈ A_α | a can be fired}
    if A*_α ≠ {} then
        a* ← arg max_{a ∈ A*_α} q_α(a)
        fire a*
    end if
end while
```

In order to properly define input and output of an action, we first introduce the notion of a *token set*. A token set contains all tokens that are read or written by an action; they are represented by their position immediately before or after firing, e.g. the second token to be read from a specific input channel or the fifth token to be written to a specific output channel during the firing. As FIFO channels only allow in-order accesses, all the token positions on a specific channel must form a sequence without any gaps in it. Formally, we define a token set as follows.

Definition 4 Let Γ be a set of channels and $\nu : \Gamma \to \mathbb{N}_0$ a function assigning each channel $\gamma \in \Gamma$ a number of tokens to be read from or written to it. Then the **token set** ψ over Γ defined by ν is a set of token positions $\psi = \{(\gamma \in \Gamma, n \in \mathbb{N}) | n \leq \nu(\gamma)\}$. $\Psi(\Gamma)$ is the set of all token sets over Γ.

To be able to formally describe functions using these tokens as input or output, we introduce the notion of a *value space*:

Definition 5 The **value space** $V(\psi)$ of a token set ψ is the set of all possible combinations of values which the tokens represented in ψ can have.

Now, we formally define an actor as a stateful computational element with a prioritised set of actions it can fire.

Definition 6 An **actor** is a tuple $\alpha = (S_\alpha, A_\alpha, q_\alpha, s^0_\alpha)$, with S_α the set of possible states of the actor, A_α a set of actions for the actor, $q_\alpha : A_\alpha \to \mathbb{N}$ a function assigning each action a *priority* and $s^0_\alpha \in S_\alpha$ the initial state.

We consider the states of an actor to be an arbitrary combination of variables of any kind. This means in particular that the state set of an actor does not have to be finite.

We define an action as follows:

Definition 7 Let α be an actor with I_α the set of its incoming and O_α the set of its outgoing channels. An **action** for this actor is a tuple $a = (r_a, w_a, f_a, g_a)$ with $r_a \in \Psi(I_\alpha)$ and $w_a \in \Psi(O_\alpha)$ the *input* and *output token sets*, $f_a : S_\alpha \times V(r_a) \to S_\alpha \times V(w_a)$ the *fire function* and $g_a : S_\alpha \times V(r_a) \to \{\text{TRUE}, \text{FALSE}\}$ the *guard function* of the action.

The action a *can be fired* if:

1. all input tokens according to r_a are available and
2. g_a evaluates to true for the current state and the available input tokens.

Upon firing, it will destructively read the input tokens (i.e. all tokens in r_a) from the channels, evaluate f_a for the current state and the tokens just read and use its return values for updating the state of α and for writing tokens to the output channels according to w_a.

Now we can define the behaviour of an actor α as an infinite repetition of the following operations: It will determine the set $A^*_\alpha \subseteq A_\alpha$ of actions that can be fired. If this set is non-empty,

Listing 1: An absolute value actor written in CAL.

```
actor Abs() int In ==> int Out: //one channel in, one out
  pos: //action: reads a token i from In and writes it to Out
    action In:[i] ==> Out:[i]
    guard i >= 0 //only fired if i is non-negative
  end
  neg: //action: reads a token i from In and writes -i to Out
    action In:[i] ==> Out:[-i]
    guard i < 0 //only fired if i is negative
  end
end
```

Listing 2: A non-deterministic merge actor written in CAL.

```
actor NDMerge() //two channels in, one out
    int InA, int InB ==> int Out:
  actionA: //reads i from InA and writes it to Out
    action InA:[i] ==> Out:[i]
  end //no guard
  actionB: //reads i from InB and writes it to Out
    action InB:[i] ==> Out:[i]
  end //no guard
end
```

the action $a \in A^*_\alpha$ with the highest priority $q_\alpha(a)$ will be fired. If there are multiple actions with the same, highest priority that can be fired, it is not defined which of those actions is fired. An illustration of this behaviour is given in Algorithm 1.

In summary, for a dataflow actor, the action which is fired (and thus the amount of tokens read and written) can depend on the state of the actor, on the value of tokens in the incoming channels and on the existence of tokens in the incoming channels. A dataflow actor must therefore be able to non-destructively read the tokens on its incoming channels (*peeking*). Furthermore, since there can be situations when the action to be fired (and thus possibly the output to be produced) depends on which token arrives first, actors can be non-determinate.

Examples for a programming language that can be described by the dataflow model given here are the CAL Actor Language [9] and its standardised variant RVC-CAL [16] [1]. Listings 1 and 2 show two code examples written in RVC-CAL. The `Abs` actor in Listing 1 has two actions `pos` and `neg`. Both read one token from the input channel `In` and write one token to the output channel `Out`. The guard expressions $i \geq 0$ and $i < 0$, respectively, ensure that, depending on the value of the token at the FIFO head of `In`, only one of both actions can fire. This actor is determinate. For comparison, the two actions of the `NDMerge` actor in Listing 2 have no guards specified, i.e. their guard function always evaluates to true. Therefore, `actionA` and `actionB` can fire whenever a token is available at `InA` and `InB`, respectively, forwarding this token to `Out`.

2.3 Problem Statement

In the following, we will define the problem to be solved in this work. To this end, we first define when an actor and a process can be regarded as equivalent.

Definition 8 Let α be a dataflow actor according to Definition 6 and π be a Kahn process as specified in Definition 1. α and π are **functionally equivalent** iff for any equal input history, α and π always produce equal output histories.

[1]The model and (RVC-)CAL differ in that (RVC-)CAL does not require the specification of priorities. Also, the latter are specified as partial orders, i.e. one only defines e.g. $q(a_1) > q(a_2)$. The slightly simpler notation we chose does, however, cover all the cases needed for this paper, since multiple actions can still have the same priority number.

Listing 3: Example for a determinate, but not KPN compatible actor.

```
actor DeterminateNotKPN()
    int InA, int InB, int InC ==> int Out:
  actionA: action InA:[a], InB:[b] ==> Out:[a]
           guard a>0  && b>0  end
  actionB: action InB:[b], InC:[c] ==> Out:[b]
           guard b<=0 && c>0  end
  actionC: action InA:[a], InC:[c] ==> Out:[c]
           guard a<=0 && c<=0 end
  actionX: action InA:[a], InB:[b], InC:[c] ==>
           guard (a<=0 && b>0 && c>0) ||
                 (a>0 && b<=0 && c<=0) end
end
```

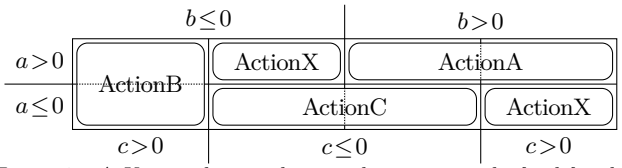

Figure 1: A Karnaugh map showing the actions to be fired for the actor from Listing 3, depending on the input token combination.

The problem regarded in this work can now be formulated as follows: *Given a dataflow actor α. Is there a functionally equivalent Kahn process π and, if so, how can it be constructed?*

The following examples shall illustrate the complexity of the problem. Of course, the actor in Listing 1 is KPN compatible while the actor in Listing 2 is non-determinate and thus clearly not KPN compatible. However, things are more complicated for the actor shown in Listing 3. The Karnaugh map in Fig. 1 shows the different actions to be fired for each combination (`a`, `b`, `c`) of the first tokens to be read from each of the input channels (provided that they exist). Since none of the guards overlap in the diagram, the action to be fired can be clearly determined from the *values* of the tokens and does *not* depend on *priorities* or the *availability* of tokens. The behaviour of the actor is thus determinate. Although some actions can still be fired if one of the channels is empty, this will only happen if the respective action would also be fired if the channel was filled.

This feature, however, cannot be achieved with KPN: A Kahn process would have to choose one channel to read from without knowing about the availability of tokens. If, for instance, this channel was `InA`, the process would block on an infinite sequence of negative integers on `InB` and of positive integers on `InC` if `InA` remained empty. This, however, is not the behaviour of the given actor. In other words, this actor is determinate but KPN incompatible. In this example, one can also graphically interpret this constellation such that it is impossible to split the Karnaugh map at the borders for `a`, `b` or `c` without cutting through one of the action guards.

This example shows that the problem regarded in this work is not the same as the problem of classifying an actor as time-dependent or not, which was discussed, e.g., in [20].

3. TRANSLATING DATAFLOW ACTORS TO KAHN PROCESSES

In this section, we discuss the translation of dataflow actors to Kahn processes. While the two-step procedure we propose consists of a KPN compatibility evaluation and a subsequent Kahn process construction, our compatibility analysis method is constructive and thus works the other way round: We first construct a Kahn process that imitates the dataflow actor's behaviour. Af-

Algorithm 2: Template for a Kahn process translation π of a dataflow actor α. The template uses initial state $s_\pi^0 = s_\alpha^0$ and an action set $A_\pi = A_\alpha$.

```
s ← s_π^0
while TRUE do
    // Find next action to be fired
    a ← SelectNextAction(s, A_π)
    // Fire the action
    in = ReadInputTokens(r_a)
    (s, out) ← f_a(s, in)
    WriteOutputTokens(w_a, out)
end while
```

terwards, if we can show that this process is functionally equivalent to the actor, we have proved the actor's KPN compatibility.

Obviously, this approach cannot detect all cases of KPN compatibility (that problem is undecidable [22]). However, we will show in the next section, using a state-of-the-art dataflow benchmark suite with 381 actors, that the method works for a large subset of KPN compatible dataflow actors in real applications.

The section continues by showing how to build the mentioned Kahn process imitation of a given dataflow actor. We will give criteria as to which requirements have to be met to guarantee functional equivalence. Afterwards, we will show a formalisation of this in which we statically analyse if these requirements are met under all possible circumstances. That would be a sufficient condition for KPN compatibility. Finally, we will discuss different translation implementations and compare the efficiency of the produced code.

3.1 Constructing a Kahn Process from a Dataflow Actor

In the following, we propose a method to build a Kahn process imitating the functionality of a given actor. The difficulty is that in contrast to the dataflow actor, the constructed Kahn process can only access the input channels using blocking, destructive reads.

We propose to construct the process using the template shown in Algorithm 2. The behaviour of the process can be described as an endless loop, each iteration consisting of two operations: Finding the next action to fire and actually firing it. As firing a dataflow action can be done natively in a Kahn process, the only difficulty lies in finding the correct action to fire, i.e. in determining the function SelectNextAction. The following theorem shows that functional equivalence between such a Kahn process and a dataflow actor can be attained if for any input, the sequence of actions fired by the process is same as with the actor.

Theorem 1 *Let α be a dataflow actor and let π be a Kahn process constructed according to the template given in Algorithm 2. π is functionally equivalent to α if for any input history, π and α always fire the same actions in the same order.*

Proof From Definition 8, we know that π and α are functionally equivalent if both generate the same output history for any given input history. According to the process template, output is only produced when firing actions. Therefore, π and α are functionally equivalent if they always fire the same actions on the same input sequence. □

The challenge is now to be able at any moment to determine which action the actor will fire without knowing about the availability of tokens or their content. In general, this is not always possible for dataflow actors where, for instance, actions can be fired or not depending on the availability of tokens or their values. However, for a large group of actors, there are possibilities of exploiting certain properties of the action guards.

- In general, guard functions do not depend on the full input token set of the associated action. In particular, many guard functions only depend on the state of the actor and can thus be evaluated without reading any tokens.
- Actions may have common input tokens. The `Abs` actor in Listing 1 shows a typical example of this setup: There are two different actions, both with guards that peek an input token. However, both of these guards peek the same input token, and one of these two actions must fire next. Consequently, the token will be read in any case and can thus be *prefetched*. After reading it, the process can decide which action to fire and pass the token on to it.
- Oftentimes, the return values of guards can be predicted without knowing all of the required input tokens. If, for instance, a guard function is a boolean AND combination of multiple terms, the result will always be FALSE if only one of these terms evaluates to FALSE. In this case, the input tokens for the other terms are not required to know that the guard is not met.

These ideas can now be combined to an algorithm that determines the action to be fired next, i.e. an implementation of SelectNextAction in Algorithm 2. For this, we assume that for each guard g, there is function predict$\langle g \rangle$, which, provided with the state of the actor and the values of the input tokens prefetched so far, evaluates to TRUE, FALSE or UNKNOWN. The following theorems shall provide a theoretical basis for the operation of the SelectNextAction algorithm.

First, we show that if the guard of a given action within a dataflow actor can be predicted to FALSE, this action will not be fired next, independently of any additional input tokens that may arrive.

Theorem 2 *Let α be an actor and $a \in A_\alpha$ be an action, with* predict$\langle g_a \rangle$ *evaluating to* FALSE. *Then a will not be the next action fired by α.*

Proof If predict$\langle g_a \rangle$ (and thus g_a) evaluates to FALSE, a cannot be fired. The return value of g_a depends on the state of the actor and on certain input tokens. Both can only be changed when firing an action. So another action has to be fired first before g_a can evaluate to TRUE. □

Now we will analyse which tokens the constructed Kahn process can prefetch at a given moment without losing functional equivalence to the original dataflow actor. The difficulty here is to avoid additional blocking which is not there in the original actor. Such a blocking could be induced by a (blocking) read operation in the Kahn process.

Theorem 3 *Let α be an actor to be translated to a Kahn process π. Let $A \subseteq A_\alpha$ be a set containing all actions the guards of which are predicted to* TRUE *or to* UNKNOWN. *Then π can prefetch all tokens from the* prefetch token set $\bigcap_{a \in A} r_a$ *of A without losing the functional equivalence to α.*

Proof According to Theorem 2, only elements of A are eligible to be fired next. Each of these actions $a \in A$ needs all tokens out of its input token set r_a before it can be fired. Therefore, α will not fire any action until those tokens which are contained in all of these input token sets have been fetched. A Kahn process

Table 1: Example actions of an example dataflow actor with the state $s \in \mathbb{N}$ and the input channels X and Y. $X[0]$ is the first token the actor receives when reading from X, $X[1]$ the second token etc. The actions' priorities increase with increasing numbers.

Action	Input tokens	Guard	Priority
a_1	—	$s=1$	1
a_2	$X[0],X[1],Y[0]$	$s=2 \wedge X[1]>0 \wedge Y[0]>0$	2
a_3	$X[0]$	$s=2 \wedge X[0]=1$	3
a_4	$X[0],X[1],Y[0]$	$s=2 \wedge X[0]=2 \wedge X[1] \cdot Y[0] \leq 0$	4
a_5	—	$s=3$	5
a_6	$Y[0]$	$s=3 \wedge Y[0]<0$	6

prefetching any token from the prefetch token set will thus never be blocked for longer than α will. When in control again, it can continue imitating α. □

With these prerequisites, we can now describe a possible implementation of SELECTNEXTACTION for the imitation of an actor α. The algorithm performs an iterative reduction of a set A of actions and can be summarised by the following steps:

0. Start with $A = A_\alpha$.
1. Prefetch all tokens from the prefetch token set of A.
2. If the guard of an action from A is predicted to FALSE considering all prefetched tokens, remove the action from A.
3. Iterate steps 1 and 2 until convergence. (Since A can only shrink, convergence is guaranteed.)

Once the iteration has converged, there are two possibilities: If all input tokens of the action in A with the highest priority have been prefetched, this action is to be fired next. Otherwise, the next action to be fired cannot be determined using this method.

We will illustrate this algorithm for the example actor given in Table 1. The actor has a rather simple state $s \in \mathbb{N}$ and two input channels, X and Y. Each of the actions a_x has a different priority $q(a_x) = x$ (this is not necessarily the case in practice). Assuming that $s=2$ and the input tokens on both X and Y are 1,2,3,4,5,..., the algorithm would behave as follows:

- Initialisation: $A = \{a_1,...,a_6\}$. No tokens to prefetch. Since $s=2$, a_1, a_5 and a_6 can be eliminated (i.e. removed from A).
- First iteration: $A = \{a_2, a_3, a_4\}$. Prefetch $X[0]$. Since $X[0]=1$, a_4 can be eliminated.
- Second iteration: $A = \{a_2, a_3\}$. No further tokens can be prefetched. No further eliminations are possible.

In this case, the algorithm stops with $A = \{a_2, a_3\}$. Since all input tokens of a_3 have been prefetched and its guard evaluates to TRUE, a_3 can be fired. a_2 is also still a candidate but cannot be fired yet because some of its input tokens are still missing. Since, however, a_3 has the higher priority and can be fired, the actor will fire a_3, independently of a_2. Thus, in this case the algorithm is able to determine a_3 as the next action to be fired. For other actor states or inputs, however, the situation may be different. Clearly, the constructed Kahn process is only functionally equivalent to the original actor if the next action to fire can be determined for *any* actor state and input. This is expressed in the following theorem.

Theorem 4 *Let α be an actor to be translated to a Kahn process π using the method described above. α and π are functionally equivalent if the implementation of* SELECTNEXTACTION *is able to determine the next action to fire for any state of the actor and of the input channels.*

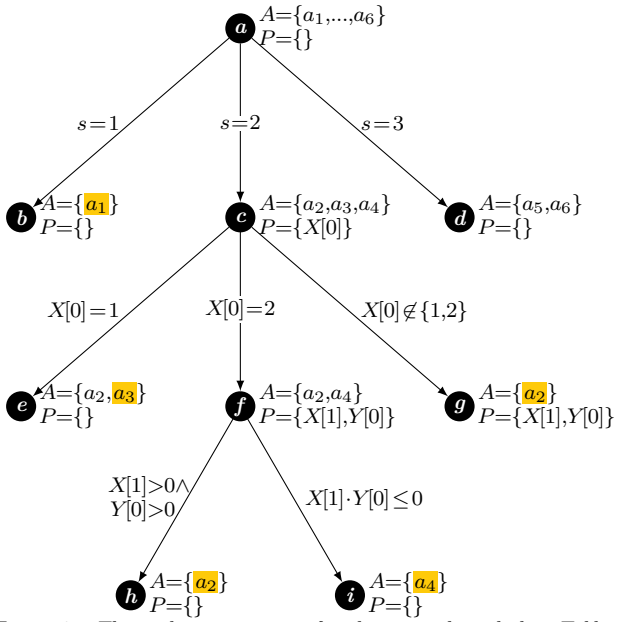

Figure 2: The peek sequence tree for the actor described in Table 1. The action to be fired in each case is highlighted.

PROOF The correctness of the operations applied by the algorithm has been proven in Theorems 2 and 3. Therefore, the set A^*_α of actions that can be fired at a given moment is a subset of the set A of candidates obtained by the algorithm. If the action in A with the highest priority can be fired, it is (i) an element of A^*_α and (ii) the action with the highest priority in A^*_α, since $A^*_\alpha \subseteq A$. π will thus fire the same action as α.

If this method works for any state and input combination, π and α will always fire the same actions and are therefore functionally equivalent, which follows from Theorem 1. □

3.2 Classification of Dataflow Actors

So far, we have seen a technique to construct a Kahn process from a dataflow actor. It tries to determine at runtime the next action to be fired. Only if this always succeeds, a correct translation was obtained and the actor can be shown to be KPN compatible. In the following, we present a static analysis method that determines if this holds true by systematically checking all possible outcomes of the SELECTNEXTACTION function introduced previously.

To this end, we construct a tree that contains all possible operations SELECTNEXTACTION might perform, i.e. reducing the set of firing candidates and prefetching tokens, depending on certain conditions that can be fulfilled or not. As this tree gives information about which tokens are fetched in which order, we call it the *peek sequence tree* (PST); a more formal definition is given later on.

For the example actor from Table 1, which was discussed above, the PST is given in Fig. 2. Every node represents a possible iteration of SELECTNEXTACTION, with the set A of firing candidates and the set P of tokens to be prefetched. The root node (marked as ***a***) represents the initialisation step. The outgoing edges of each node (i.e. those leading further away from the root node) represent the different possibilities of how the SELECT-

NEXTACTION may proceed, leading to the next iteration step in the respective cases. They are annotated with a condition to be met such that the edge is followed. Since the prefetch token set of the root node is empty, the conditions leading away from it only contain the actor state s. The edge annotations further down will also have conditions concerning the prefetched tokens. Note that all these conditions are mutually exclusive for edges leaving the same node. However, they need not cover all possible cases, but only those which are covered by the actions of the original actor.

The leaves of the tree are equivalent to all possible outcomes of SELECTNEXTACTION:

- The iteration scenario discussed in Section 3.1 is represented by the path $\boldsymbol{a} - \boldsymbol{c} - \boldsymbol{e}$.
- Node $\boldsymbol{b}$ is a very straightforward case in which the action to fire is determined only by the state of the actor.
- Nodes $\boldsymbol{h}$, $\boldsymbol{i}$ and $\boldsymbol{g}$ represent cases in which, due to repeated token prefetching and firing candidate elimination, only one action to fire is left.
- Finally, in the case of node $\boldsymbol{d}$, the action to fire cannot be determined. This is because action a_6 has a higher priority than a_5, but also needs more input tokens. The actor would thus fire a_6 if these tokens are available and a_5 otherwise. The example actor regarded here is thus *not* KPN compatible.
- Another possible case, which does not occur in this example, is that of an ambiguous actor specification. An actor is specified ambiguously if it has two actions with the same priority, without mutually exclusive guards and if the input token set of the one action is a subset of that of the other action. In the PST, this would lead to a leaf with multiple actions of the same priority. One possible way of handling this issue would be to arbitrarily give priority to one of the actions. This is done in many backends as well as in [20]. In this work, however, since our final goal is translation to a KPN process, we choose a conservative approach and do not classify the actor as KPN compatible in order to prevent a translation when the actor semantics as intended by the programmer are not clear.

In the following, we will give the formal definition of a PST and we will show how to construct it. To this end, we first discuss how the predict$\langle\cdot\rangle$ function introduced earlier can be implemented. We do so by assuming that every guard is a boolean AND combination of multiple terms referred to as *constraints*. According to our following definition, a constraint requires a set of tokens (the *peek tokens*) in order to be evaluated and it can be met or not, according to a boolean function:

Definition 9 Let I_α be a set of input channels and S_α a set of possible states of an actor α. A **constraint** to an action for α is a tuple $c=(p_c,e_c)$ with $p_c \in \Psi(I_\alpha)$ a token set of *peek tokens* and $e_c : S_\alpha \times V(p_c) \to \{\text{TRUE},\text{FALSE}\}$ the *evaluation function*.

The negation of a constraint c is given as $\overline{c} := (p_c, \text{not}(e_c))$. Likewise, the combination of two constraints c and d is given as $c \wedge d := (p_c \cup p_d,\ e_c \wedge e_d)$.

With this definition, a guard can be expressed as the boolean AND combination of all elements in a set of constraints. Therefore, we can assign each action such a set representing its guard. To be able to evaluate as many constraints as early as possible and thus maybe to eliminate certain actions as firing candidates early on, we would like to break down each guard into as many constraints with as small peek token sets as possible.

Definition 10 Let a be an action. The **constraint set** C_a of a is a set of constraints such that

$$\bigwedge_{c \in C_a} c \ = \ (p, g_a),\ \ p \subseteq r_a$$

and that for each constraint $c \in C_a$, there are no two constraints c', $c'' \neq c$ such that $c' \wedge c'' = c$.

In the example actor described in Table 1, the guard of action a_4 can be decomposed to the constraint set

$$C_{a_4} = \{\ (\{\},\ s=2),\quad (\{X[0]\},\ X[0]=2),\quad (\{X[0],X[1],Y[0]\},\ X[1]\cdot Y[0] \leq 0)\ \ \}.$$

Our definition of a PST is now as follows:

Definition 11 A **peek sequence tree (PST)** for an actor α is a tree $T=(N,L)$ on which each node $n \in N$ is annotated with a set of actions $A(n) \subseteq A_\alpha$ and each edge $l \in L$ is annotated with a constraint $c(l)$.

We define the prefetch token set of a node according to Theorem 3:

Definition 12 Let n be a node in a PST. Then its **prefetch token set** is $P(n) = \bigcap_{a \in A(n)} r_a$.

A PST $T=(N,L)$ must fulfil the following conditions: For each edge $l \in L$ with $n \in N$ being its parent (source) node, it must hold that $p_{c(l)} \subseteq P(n)$. For any two edges $l,m \in L$ with a common parent node, it must hold that $c(l)$ and $c(m)$ cannot be satisfied at the same time, i.e. $e_{c(l) \wedge c(m)} \equiv \text{FALSE}$.

The rest of this section describes the construction of a PST for an actor α. This procedure can be formalised as follows: A root node n_0 is created with $A(n_0) = A_\alpha$. For each action $a \in A(n_0)$, the set of evaluable constraints $C'_a = \{c \in C_a | p_c \subseteq P(n_0)\}$ is determined and combined to the *strictest evaluable constraint* $c'_a = \bigwedge_{c \in C'_a} c$, i.e. the largest top-level sub-expression of g_a that can already be evaluated with the tokens available through prefetching. (If C'_a is empty, we have $c'_a = (\{\},\text{TRUE})$.) For action a_4 from the example actor, the strictest evaluable constraint for the root node is $c'_{a_4} = (\{\}, s=2)$.

From these $k := |A(n_0)|$ strictest evaluable constraints, each one can theoretically be met or not, which in total gives 2^k cases. These cases can be expressed as a set of constraint combinations

$$C_{\text{theo}}(n_0) := \left\{ \bigwedge_{a \in A(n_0)} x_a \ \middle|\ x_a \in \{c'_a, \overline{c'_a}\} \right\}.$$

For the example actor, the possible cases for the root node are $s=1 \wedge s=2 \wedge s=2 \wedge ...$, $\overline{s=1} \wedge s=2 \wedge s=2 \wedge ...$, $s=1 \wedge \overline{s=2} \wedge s=2 \wedge ...$ etc.

As constraints are often related and some contradict each other, not all of the combinations are satisfiable, as clearly seen in the example. Using a satisfiability modulo theories (SMT) solver, which is also provided with the set S_α of possible states of α, one can eliminate the unsatisfiable combinations. Also the case that none of the constraints is met can be eliminated, since this case is not covered either in the original actor. In the example, all of the $2^6 = 64$ possibilities are eliminated except for three:

- $s=1 \wedge \overline{s=2} \wedge \overline{s=2} \wedge \overline{s=2} \wedge \overline{s=3} \wedge \overline{s=3}$
- $\overline{s=1} \wedge s=2 \wedge s=2 \wedge s=2 \wedge \overline{s=3} \wedge \overline{s=3}$
- $\overline{s=1} \wedge \overline{s=2} \wedge \overline{s=2} \wedge \overline{s=2} \wedge s=3 \wedge s=3$

These combinations obviously simplify to $s=1$, $s=2$ and $s=3$, which have been noted down in Fig. 2. In the real implementation, the number of satisfiability evaluations can be reduced by applying various optimisations that shall not be discussed here.

Table 2: Satisfiability calculations for the PST in Fig. 2 at node $\boldsymbol{c}$

Actions	Constraint combination	Eliminate?
$\{\}$	$s=2 \wedge \overline{\text{TRUE}} \wedge \overline{X[0]=1} \wedge \overline{X[0]=2}$	Yes, empty set
$\{a_2\}$	$s=2 \wedge \text{TRUE} \wedge \overline{X[0]=1} \wedge \overline{X[0]=2}$	**No, retain**
$\{a_3\}$	$s=2 \wedge \overline{\text{TRUE}} \wedge X[0]=1 \wedge \overline{X[0]=2}$	Yes, unsatisfiable
$\{a_2,a_3\}$	$s=2 \wedge \text{TRUE} \wedge X[0]=1 \wedge \overline{X[0]=2}$	**No, retain**
$\{a_4\}$	$s=2 \wedge \overline{\text{TRUE}} \wedge \overline{X[0]=1} \wedge X[0]=2$	Yes, unsatisfiable
$\{a_2,a_4\}$	$s=2 \wedge \text{TRUE} \wedge \overline{X[0]=1} \wedge X[0]=2$	**No, retain**
$\{a_3,a_4\}$	$s=2 \wedge \overline{\text{TRUE}} \wedge X[0]=1 \wedge X[0]=2$	Yes, unsatisfiable
$\{a_2,a_3,a_4\}$	$s=2 \wedge \text{TRUE} \wedge X[0]=1 \wedge X[0]=2$	Yes, unsatisfiable

For each of the cases that have not been eliminated, a child node is inserted. The edge to it is annotated with the constraint combination corresponding to the case. The child node itself is annotated with the set of all actions the strictest evaluable constraint of which was assumed to be met in the constraint combination. See Fig. 2 for the example actor.

For all the child nodes, the same procedure is carried out recursively. However, only constraints that could not be evaluated before are regarded now. The old constraints are taken into account by combining all constraints of the edges that lead from the root node to the current node and by adding this combination as an additional constraint for the SMT solver. Table 2 shows the procedure for node $\boldsymbol{c}$ in the PST for the example actor. The constraint "inherited" from above is $s=2$; it is therefore only added at the beginning of each combination.

As soon as only one child node would be inserted for a node, the recursion is stopped.

We can upper bound the complexity of the proposed algorithm to construct the PST:

Theorem 5 *For an actor with k actions, the maximum number of nodes in the PST is smaller than* $2^{\frac{1}{2}(k^2+k)}$.

PROOF If a child node has the same number of actions as its parents, no progress is made and the recursion is stopped. Therefore, a child has in the worst case one action less than its parent. In the worst case, the root node can have up to 2^k-1 child nodes. Each of these child nodes can then have up to $2^{k-1}-1$ children, which again can have $2^{k-2}-1$ children each and so forth. Multiplying these numbers, one obtains the maximum number of leaves in the tree. Also counting the non-leaf nodes, one has $1+(2^k-1)\cdot\left(1+(2^{k-1}-1)\cdot(1+...)\right) < (1+2^k-1)\cdot(1+2^{k-1}-1)\cdot... = 2^k\cdot 2^{k-1}\cdots 2^0 = 2^{0+1+2+...+k} = 2^{\frac{1}{2}(k^2+k)}$. □

Note that this is the upper bound for pathological cases. In our experimental evaluations with 381 real actors, the number of nodes stayed way below it in each case. Section 4 will show that even for large actors, the tree can be constructed in an acceptable time frame in spite of its theoretically exponential complexity. In extreme cases, one could stop the PST construction prematurely without a classification result.

3.3 Constructing the translated Kahn process

With the results from the classification problem in mind, the solution to the translation problem is straightforward. If an actor has been classified as KPN compatible, one just needs to construct a process as described in Section 3.1.

One can simply implement the SELECTNEXTACTION function as shown there for determining the action to fire. This has the advantage that the complexity of this algorithm is polynomial with respect to the number of actions.

In practice, however, more lightweight code can be generated directly following the structure of the PST constructed during classification. For each node, prefetching code needs to be produced whereas each edge in the tree will be a branch in the code. Like this, dynamic predictions of guards can be replaced by simple, hard-coded if statements.

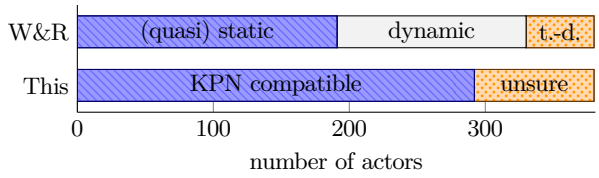

Figure 3: Overall comparison of the classification results of the algorithm of Wipliez and Raulet ("W&R') and that proposed in this paper("this"). The abbreviation "t.-d." stands for time-dependent; "unsure" designates potentially KPN incompatible actors.

4. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of the proposed classification and translation algorithm using a state-of-the-art dataflow benchmark suite. The goal is to answer the following questions: *a*) What percentage of realistic RVC-CAL actors does the proposed algorithm classify as KPN compatible? *b*) What are the reasons for KPN compatible actors not being classified as such? *c*) Does the proposed translation of KPN compatible dataflow actors into Kahn processes indeed improve the performance of streaming applications?

4.1 Experimental Setup

The proposed classification and translation algorithm has been implemented as an extension to the Open RVC-Cal Compiler (ORCC) [21] using the z3 SMT solver [8]. The corresponding ORCC benchmark suite [1] contains a total of 549 CAL actors. In order to provide meaningful data, current research projects (i.e. immature work under construction) as well as overly simplistic actors such as "hello world" examples have been left out from the evaluation. With the exception of those, the proposed classification algorithm has been tested on all available actors, 381 in total. In particular, the set of applications contains various video decoders (H.265 part2, H.264 PHiP, H.264 CBP and MPEG-4 SP, AVS), the JPEG and JPEG2000 image compression codecs, for telecommunications the ZigBee transmitter baseband description and a digital predistortion filter [11], a set of basic digital filters, a cryptographic library, a WAV audio player, a GZIP decompressor and implementations of several CHSTONE benchmark suite applications.

The classification algorithm discussed above was run on an Intel Core i5-3210M processor, as a single threaded implementation. The classification of all 114 actors of the H.264 PHiP decoder, one of the most elaborate dataflow applications in the benchmark suite and with several highly complex actors, took about 65 seconds.

4.2 Comparison to Other Classifiers

In the following, we evaluate the performance of the proposed classification algorithm. To this end, we first regard the counts of the different classification results for the algorithm proposed in this work and for the algorithm by Wipliez and Raulet (W&R) [20]. These numbers are given in Fig. 3.

For W&R, the group *(quasi) static* combines the three possible results SDF [13], CSDF [4] and quasi-static [5], which are all KPN compatible by construction.

Actors are marked as *time-dependent* by the W&R classifier if it finds constellations similar to that in node ***d*** in Fig. 2. As explained in Section 3.2, such actors are KPN incompatible. Note, however, that time-dependency is *not* the same as non-determinacy; it is only a necessary condition for the latter.

Finally, all other actors are classified as *dynamic*, i.e. determinate but not (quasi) static. These actors may or may not be KPN compatible.

The two classifiers regarded here cannot be compared directly for two reasons: Firstly, the different classification categories and secondly, their different treatment of ambiguous actor specifications as described in Section 3.2. The W&R classifier enforces (arbitrarily) a total priority ordering of all actions, which, in the extreme case, leads to an actor being classified either as time-dependent or as quasi static, depending on the order of the action specifications in the source code. The classifier we propose always classifies actors with ambiguous specifications as potentially KPN incompatible.

Consequently, we have to look at the cases in closer detail:

- The results of the (quasi) static group of W&R can be confirmed by our algorithm in so far as it classifies all of the concerned actors as KPN compatible. The only exception is given by three ambiguously specified actors.
- The W&R classification of actors as time-dependent uses similar criteria to those in the algorithm we propose. Accordingly, none of the actors classified as time-dependent was classified as KPN compatible by the algorithm proposed in this work. However, manual classification showed that more than a quarter of these actors are KPN compatible, which was not recognised by the algorithms. The reasons of W&R time-dependent misclassifications and their frequency are similar to those for non-classifications in our algorithm, which will be discussed later on in detail.
- Out of the actors classified as dynamic by W&R, 75 % were classified as KPN compatible. Another 7 % of these actors is KPN compatible as well but were not recognised as such. Note that these rates do not differ significantly from the overall KPN classification rate of 77 % with additional 6 % not recognised. In other words, for the regarded set of actors a W&R classification as dynamic does not provide information about the KPN compatibility of an actor.

In summary, the W&R classifier can – leaving aside the ambiguously specified actors – classify 185 out of 363 actors (or 51 %) with certainty as KPN compatible, whereas the algorithm proposed in this work can do the same with 292 actors (or 80 %). The number of recognised KPN compatible actors is thus 58 % higher.

4.3 Comparison to Manual Classification

In addition to the comparison with other classifiers, we also investigated on an absolute scale the classification quality of the algorithm proposed in this work. To this end, we undertook a manual classification effort of all the actors in the set.

Since the algorithm we propose guarantees KPN compatibility for all actors classified accordingly, we did not cross-check all of these actors manually, but we took samples at random and were able to validate the correctness of the algorithm and its implementation.

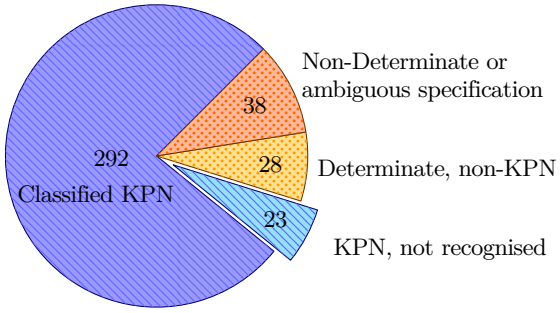

Figure 4: Manual classification results

Listing 4: Simplified example for a KPN compatible actor not recognised as such.

```
bool ax:=false, bx:=false, cx:=false;
a: action => guard         !ax do ax := true; end
b: action => guard ax and !bx do bx := true; end
c: action => guard bx and !cx do cx := true; end
```

All actors which the algorithm did not classify as KPN compatible were checked manually with the help of its output. Figure 4 shows the results of this manual classification effort. While 292 of the 381 actors were correctly classified as KPN compatible, there are 23 more KPN compatible actors, which, however, were not recognised as such. 66 actors are not KPN compatible for various reasons, which will be discussed in Section 4.4. In summary, our classifier in 94 % of the cases obtained the same result as an ideal classifier would. The optimisation possibilities for the other 23 cases will be discussed below. Note that we classified all actors *on the basis of the given implementation*, not on the basis of whether there could exist a KPN compatible implementation of their functionality.

During manual classification, we analysed why KPN compatible actors were not recognised as such. With the exception of one actor, which caused an error of the SMT solver, the reason was always the determination of the actors' state sets (S_α for an actor α). The implementation we used for the experiments is very simplistic: It assumes all combinations of all possible values of the actor's state variables to be the set of states the actor can have instead of analysing the actions to find out which values and which combinations thereof can actually be attained.

Listing 4 shows a simplified, but realistic example, in which there are three actions a, b and c, each of which during execution sets a state variable to true to indicate it has been fired. Their guards ensure no action is fired twice and each action is only fired after the one above it. Thus, actions a, b and c are fired in exactly that order. The classifier will, however, notice that the guards of actions a and c are not mutually exclusive since it cannot establish a link between the three state variables. It will thus conclude that these two actions can be fired at the same time and it will assume an ambiguous specification. Intelligent state analysis, however, would yield that the combination `ax=false and bx=true` is not contained in the state set of the actor. Using this information, the guards of actions a and c could be recognised as mutually exclusive and the actor as KPN compatible.

4.4 Further results of manual classification

The manual classification also provided results concerning the nature of the KPN incompatible actors. Although these results do not affect the classification performance of the algorithm proposed in this work, they can give hints about how it might

Listing 5: Simplified example for an unintentionally KPN incompatible actor.

```
actor Sum()
    int DataIn, bool EndOfStream ==> int SumOut:
  int sum := 0;
  readData: action DataI:[i] ==>
    do sum := sum + i;
  end
  done: action EndOfStream:[eos] ==> SumOut:[sum] end
  priority
    readData > done
  end
end
```

be used to aid programmers in writing actors or about which other classifications might be desirable to have.

The actors analysed here can be divided into two groups: determinate but KPN incompatible actors, which always produce the same output for the same input but cannot be expressed as Kahn processes, and non-determinate or ambiguously specified actors, which may produce different output for the same input. Both groups will be discussed in the following.

The group of **determinate, yet KPN incompatible actors** is quite diversified. In addition to actors similar to that shown in Listing 3, it features two more kinds of actors.

One kind of actors performs multiple unrelated operations. These actors could, in fact, be replaced by multiple Kahn processes. A (sequential) Kahn process as defined in Definition 1, however, cannot produce this behaviour.

Another kind of actors contains two sets of actions: The first set describes the actor's main behaviour and is completely KPN compatible. In parallel to it, the second set has the task of pre-buffering input tokens in internal buffers of the actor. This is a low-level optimisation with the idea that if the actor cannot perform the main calculations, it can still use the time for pre-buffering data. While the order and the firing counts of the actions thus vary, the output is still always the same and these actors are determinate.

Non-determinate or ambiguously specified actors may produce different output for the same input. However, the two groups differ in one point: While ambiguous specifications should clearly be avoided, non-determinacy is sometimes necessary, for instance in the case of a video streaming application which has to react to video input not arriving within a certain deadline.

From the semantics of each of the 20 non-determinate actors amongst the actors regarded in this evaluation, however, it can be concluded that non-determinacy in these cases is unintended. This meets the fact that all of the applications (video decoders, cryptographic applications etc.) in the set are supposed to be determinate.

The possible programming mistakes we identified amongst non-determinate and ambiguously specified actors are often the same. In most cases, it seems the author of the concerned actor did not realise that two guards actually overlap each other. In the case of ambiguity he may also have forgotten to specify a higher priority for one of the actions. The fact that most backends in such cases typically fire the action which comes first in the source code leads unfortunately often to this kind of error not being discovered.

In other cases assumptions about the input are made, usually founded on the concrete data sent in a particular graph. However, the behaviour of an actor is clearly defined only if it is unambiguous for *any* input.

Table 3: Execution time of CAL actors when being scheduled either using the default scheduler of ORCC or as a Kahn process.

actor	platform	scheduler		speed-up
		ORCC	Kahn	
mvseq	DSP	156 691 cycles	85 249 cycles	1.84 x
invpred	DSP	129 641 cycles	83 478 cycles	1.55 x
mvseq	RISC	241 544 cycles	151 636 cycles	1.59 x
invpred	RISC	453 836 cycles	230 188 cycles	1.97 x

In the case of non-determinacy, we found another pattern, which is illustrated in Listing 5. This actor reads data on one channel and is informed about the end of the data stream on a second channel (typically, both channels come from the same process). While this apparently worked well in the tests of the programmers, wrong output would be produced if the data channel delayed the tokens for longer than the end-of-stream channel, such that the end-of-stream token arrived before the last data tokens. This situation could be avoided if the data channel supported the transmission of special control tokens. In this case, the second channel would not be necessary and the actor would actually be KPN compatible.

All these results show that KPN-incompatibility is often unintended. Especially for larger actors (there are several with more than 2000 lines of code), a KPN compatibility analysis, as performed by the algorithm presented here, may thus prove to be a valuable tool for a programmer, even if he does not target a KPN implementation of his actors.

4.5 Performance of a Dataflow Actor and a KPN Process

Next, we evaluate if the proposed translation of a KPN compatible actor to a Kahn process can be used to improve the performance dataflow graphs. To this end, C versions of the translated Kahn processes were generated as described in Section 3.3 and compared to C code generated conventionally by ORCC [21].

The code was compiled and then run on two systems:

- A Texas Instruments TMS320C6416 Fixed Point DSP featuring L1 instruction and data caches of 16 KB each. The evaluation was done on the Texas Instruments cycle accurate device simulator, which takes account of cache behavior. The CCS IDE version was 5.5.0.
- An Altera Nios II/f RISC processor with 4 KB L1 instruction and 2 KB L1 data cache. The evaluation was done by synthesizing the processor core on an Altera Stratix III FPGA and by measuring the cycle time with the SignalTap II logic analyzer. The Quartus II software version was 13.1.

The measurements were performed with two different actors: The "Mgnt_MVSequence_LeftAndTopAndTopRight" actor (mvseq) and the "Algo_DCRInvPred_LUMA_16x16" actor (invpred), both from the MPEG-4 Part 2 Simple Profile decoder. The former consists of 7, the latter of 10 actions. The achieved results are summarized in Table 3.

For these actors, a speed-up between 1.55 x and 1.97 x was achieved. The reason for these improvements is that, instead of linearly iterating over all actions like CAL implementations, the KPN translation follows the structure of the PST, i.e. it performs a sort of binary search for the next action to be fired. It also does not need to check the availability of tokens.

Of course, the influence of this overhead reduction decreases with a higher computational complexity of the actions to be fired. Still, the examples show its relevance in real production code.

5. RELATED WORK

Classifying dataflow actors into more restrictive dataflow models has recently been considered as an efficient technique to improve the execution performance of dataflow graphs, e.g. by reducing the number of communication channel accesses. In particular, a methodology to classify dataflow actors into actors adhering to the synchronous dataflow (SDF) [13] model and the cyclo-static dataflow (CSDF) [4] model is presented in [22]. In order to model dynamic and time-dependent behavior, each actor is described by a finite state machine that controls the communication behavior of the actor. In contrast to our work, their approach is limited to only classify static dataflow actors.

A method to classify dataflow actors into static, dynamic, and time-dependent actors is presented in [20] based on satisfiability and abstract interpretation. While this method can identify SDF, CSDF and quasi-static actors, which are KPN compatible by construction, it cannot identify more general patterns of KPN compatibility. The method for detecting time dependency could be used for showing KPN incompatibility, but is somewhat inaccurate as seen in Section 4. With its ability to identify (quasi) static actors, it can, like [22], be regarded as a complement to our approach.

In [17], a scheduling approach for semi-dynamic dataflow graphs is presented. To this end, a novel dataflow model is introduced that constructs actors with sets of modes representing fixed behaviors. Then, it is shown that a set of static dataflow graphs can be derived by decomposing the graph by its modes.

Other approaches trying to improve the performance of dataflow actors exist, e.g. by scheduling the actor more efficiently. For instance, the technique proposed in [6] identifies most scheduling decisions of a dynamic dataflow actor at compile time by determining most of the schedule statically. In [10], this approach has been extended to also analyze the state space of certain network partitions.

Outside the field of dataflow networks, the problems of availability of input variables and of obtaining them has been discussed as well. Among others, [3] and [19] analyse formal representations of algorithms with a particular focus on fetching variables. [2] investigates in the domain of synchronous programming whether a given module can iteratively infer the availability of all required input variables from its state (*endochrony*). All these approaches have in common that they work on program descriptions in which every input variable has to be fetched explicitly. This form of specification has natural representations as trees or graphs similar to the PST shown in this work. In dataflow networks, however, fetching *all* input variables upon firing an actor is *one atomic operation* as well as checking if an actor can fire. Breaking up these atomic operations and constructing a PST is thus less obvious than with other programming models and the actual contribution of this work.

6. CONCLUSION

In this paper, we have presented a novel algorithm to classify dataflow actors that are specified according to the CAL language into KPN compatible and potentially KPN incompatible actors. A dataflow actor is KPN compatible if it can be represented as an infinite program that only performs blocking, destructive read accesses, calculations and non-blocking write accesses. Based on the classification algorithm, we have described a formal method to translate a KPN compatible dataflow actor into a Kahn process. We have demonstrated the viability of our algorithms by implementing them in the RVC-CAL framework. In fact, the proposed classification algorithm has been capable to identify 93 % of all KPN compatible, mature actors from the ORCC benchmark suite. Finally, we have shown that the performance of KPN compatible actors can be improved by up to 1.97 x when executing them as Kahn processes instead of dataflow actors.

7. ACKNOWLEDGEMENTS

This work was supported in part by the Academy of Finland projects DORADO and UNICODE and by the UltrasoundToGo RTD project (no. 20NA21 145911), evaluated by the Swiss NSF and funded by Nano-Tera.ch with Swiss Confederation financing.

8. REFERENCES

[1] Open RVC-CAL Application Repository. `https://github.com/orcc/orc-apps`, 2014.

[2] A. Benveniste et al. Compositionality in dataflow synchronous languages. *Inf. Comput.*, 163(1):125–171, Nov. 2000.

[3] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20(3):265–321, 1982.

[4] G. Bilsen et al. Cycle-Static Dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

[5] J. Boutellier et al. Quasi-static scheduling of cal actor networks for reconfigurable video coding. *Journal of Signal Processing Systems*, 63(2):191–202, 2011.

[6] J. Boutellier, M. Raulet, and O. Silvén. Automatic Hierarchical Discovery of Quasi-Static Schedules of RVC-CAL Dataflow Programs. *Journal of Signal Processing Systems*, 71(1):35–40, 2013.

[7] J. Ceng et al. MAPS: An Integrated Framework for MPSoC Application Parallelization. In *Proc. Design Automation Conference (DAC)*, pages 754–759, 2008.

[8] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[9] J. Eker and J. Janneck. CAL Language Report. Technical report, University of California at Berkeley, Dec 2003.

[10] J. Ersfolk et al. Scheduling of Dynamic Dataflow Programs based on State Space Analysis. In *Proc. IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 1661–1664, 2012.

[11] A. Ghazi et al. Low power implementation of digital predistortion filter on a heterogeneous application specific multiprocessor. In *Proc. IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 8336–8340, May 2014.

[12] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. IFIP Congress in Information Processing*, volume 74, pages 471–475, 1974.

[13] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[14] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[15] D. B. MacQueen. Kahn networks at the dawn of functional programming. *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, pages 95–137, 2009.

[16] M. Mattavelli, I. Amer, and M. Raulet. The Reconfigurable Video Coding Standard. *IEEE Signal Processing Magazine*, 27(3):159–167, 2010.

[17] W. Plishker, N. Sane, and S. Bhattacharyya. A Generalized Scheduling Approach for Dynamic Dataflow Applications. In *Proc. Conf. on Design, Automation and Test in Europe (DATE)*, pages 111–116, 2009.

[18] L. Schor et al. Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems. In *Proc. Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 71–80, 2012.

[19] G. Winskel. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *Lecture Notes in Comp. Science*, pages 325–392. Springer, 1987.

[20] M. Wipliez and M. Raulet. Classification of Dataflow Actors with Satisfiability and Abstract Interpretation. *Int'l Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, 3(1):49–69, 2012.

[21] H. Yviquel et al. Orcc: Multimedia Development Made Easy. In *Proc. Int'l Conf. on Multimedia*, pages 863–866. ACM, 2013.

[22] C. Zebelein et al. Classification of General Data Flow Actors into Known Models of Computation. In *Proc. Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 119–128, 2008.