Modeling Control Tokens for Composition of CAL Actors

Johan Ersfolk[1,3], Ghislain Roquier[2], Johan Lilius[1], Marco Mattavelli[2]
[1]Åbo Akademi University, Finland
[2]École Polytechnique Fédérale de Lausanne, Switzerland
[3]Turku Centre for Computer Science, Finland

***Abstract*—Dataflow programming is typically used as an intuitive representation for describing multimedia and signal processing applications as computation nodes which communicate through FIFO queues. To run a dataflow network, consisting of several nodes, either run-time or compile-time scheduling is required. Compile-time scheduling techniques are typically based on token rates between nodes and for languages such as CAL, which are expressive enough to describe an actor with any behavior, run-time scheduling is needed in the general case, introducing an overhead. However, the well defined structure of dataflow programs enables analysis of the dependencies of the program and partitions with piecewise static schedules can be derived. In this paper we describe how actor partitions with control tokens can be modeled such that a correct scheduler, where most scheduling decisions are taken at compile-time, can be derived for the resulting composed actor.**

***Index Terms*—Dataflow programming, actor composition, MPEG-4 decoder**

I. Introduction

The typical applications for dataflow programming are from the domain of multimedia and signal processing. Signal processing algorithms can often be described as functional mappings between the inputs and the outputs and can be implemented with static token rates. Multimedia applications on the contrary, use various coding tools (i.e. algorithmic blocks) and allow different combinations of coding tools to be used for the processing. When implementing such applications as dataflow programs, this means that, while many coding tools such as various transforms can be implemented as synchronous dataflow (SDF) [1], a well-known static dataflow model, the programs implementing multimedia applications will include choices that must be taken at run-time. These choices are a result of the need to choose the appropriate coding tools but also, in some cases, a result of implementation choices in the dataflow program.

The RVC-CAL actor language has been standardized as the language to be used to describe the different coding tools of the MPEG Reconfigurable Video Coding (RVC) standard [2]. RVC-CAL provides the expressiveness that is needed to describe the various coding tools. The coding tools, represented by RVC-CAL actors, are assembled into a program by connecting the actors with unidirectional order preserving queues. The actors are allowed to execute in parallel and define their internal scheduling depending on the actors inputs and/or internal state. As a consequence, compile-time scheduling of an RVC-CAL program, consisting of a network of connected actors, is not a trivial task. Scheduling in an SDF like manner based on token rates at compile-time is in general not an option as the actors does not have static token rates, instead the token rates may depend on the input values of the actor or on the current state of the actor. Dynamic (run-time) scheduling introduces significant overhead from evaluating guards at run-time, for this reason, quasi-static scheduling methods, where piecewise static schedules are identified at compile-time while a minimum number of scheduling decisions are left for run-time, have been proposed [3], [4], [5].

Dynamic scheduling is needed when actors have input dependent conditions and consequently, some of the queues carry control tokens between the actors. The actors in turn can retransmit control tokens by defining how an output port depends on an input port. In this paper we focus on constructing models that capture this essential information which is needed for scheduling and also to decide on how to partition a network such that the scheduling becomes efficient. The abstract model of the control token propagation enables reasoning about whether a partition after a composition still accepts every input sequence allowed in the original program. The problem is related to the actual control values entering the partition and thereby also describes the guards required to perform the scheduling. When the models are in place, approaches such as [5] can be used to find the actual schedules.

II. Background and Related Work

A CAL program consist of a set of actors exchanging data tokens through First-In First-Out queues (FIFOs). The actors execute the program by firing eligible actions. Actions are eligible depending on the availability of input tokens, the values of the input tokens, and the internal state of the actor (inside guard statements). Each action may consume and/or produce tokens from one or more input or output ports connected to the FIFO channels; an action may also have no input or output.

The execution of a CAL dataflow program is asynchronous (i.e. it abstracts from time) and each actor can fire independently from all the others as far as one of its actions is eligible. However, sometimes the number of processing elements is lower than the number of actors. In that case, actors assigned on the same processing element must be scheduled by an external scheduler. An actor can fire an action only

if one of its actions is eligible. An action is eligible if: 1. tokens are available, 2. its guard expression (including any state predicate) evaluates to true, 3. it is enabled by the action scheduler, and 4. it has a higher priority if more than one action is enabled by the action scheduler in that state. The CAL action scheduler, if present, is a CAL language operator that expresses, in the form of finite state machine (FSM) transitions, when actions are eligible. An action is only fired if the current state has a transition corresponding to that action.

The scheduling of a CAL program is distributed on each of the actors and is described by the FSM, guards and priorities. In a software implementation, where there is a smaller number of processor cores than actors or the actors are too fine grained for the architecture, actors mapped to the same core must be scheduled.

Different quasi-static scheduling approaches, where static schedules of parts of a CAL program can be derived, have been presented. In [4] CAL actors are classified to belong to more restricted models of computations such SDF [1], CSDF [6] or PSDF [7], in order to allow more efficient scheduling. In [8] static regions, spanning over several actors, are identified and scheduled at compile-time. Other approaches such as [5] and [3], make use of the network partition state, consisting of the values of a subset of the variables or inputs to the partition, to find deterministic sequences of action firings. For such approaches it is of importance to identify the information in a network that is of relevance for the scheduling of the program in order to choose partitions and find the possible scenarios of those partitions. Another approach, presented in [9] constructs a model, called a machine model, which captures the action selection process of the actors and can be used to reduce the number of possible execution paths and remove the evaluation of unnecessary conditions. This model also enables actor composition based on token counts.

The difference in our approach is that we attempt to model how actors interchange actual control values and not only the number of tokens. The idea is to make control tokens a visible part of the model such that one or more of the scheduling approaches mentioned above can be used for the actual static scheduling.

III. Control Token Propagation

In a CAL actor, the token rates are fixed for each action. For simple actors, it is therefore possible to describe the behavior of individual actors as more restricted models of computation with predictable token rates [4]. However, the existence of control tokens makes actor composition based on token rates impossible in the general case. While the token rates of actions are explicitly defined, the propagation of control tokens must be derived from the implementation of the actions. A value on a FIFO is considered to be a control token if it will eventually end up in a guard expression; the path of this control token is analyzed backwards through the network, from the guard towards the input-stream. This search either terminates at an actor sending the value of a constant or a variable not depending on inputs, or at the input stream. In any case, this control token path describes what must be known in order to schedule the program and where this information is generated.

e	$\in$	Expressions
S	$\in$	Statements
x, y	$\in$	Variables
n	$\in$	Numerals
op	$\in$	Operators
p, q	$\in$	Ports
$id, state$	$\in$	Identifiers
e	$::=$	$x \mid n \mid e_1\ op\ e_2$
S	$::=$	x **:=** $e \mid skip \mid S_1;\ S_2$
		$\mid$ **if** $(e)\ S_1$ **else** $S_2 \mid$ **while** (e) **do** S
actor	$::=$	**actor** *id* () **...** $\Rightarrow$ **...**
		vars
		action
		schedule fsm *state* : *trans* **end**
		end
action	$::=$	**action** : *id* : *ports* $\Rightarrow$ *ports*
		vars
		guard e
		do S **end**
		end
		$\mid$ *action*; *action*
vars	$::=$	x **:=** $n \mid$ *vars*; *vars*
ports	$::=$	p**:** **[**x**]** $\mid$ *ports*; *ports*
trans	$::=$	*state* **(** *id* **)** $\rightarrow$ *state*
		$\mid$ *trans*; *trans*

TABLE I
THE BASIC SYNTAX OF AN ACTOR DECLARATION

Network	$=$	$(Actors, Ports, \chi)$
p, q	$\in$	$P_i \cup P_o$
χ	$:$	$P_i \nrightarrow P_o$

TABLE II
NETWORK STRUCTURE

In [10] a method for identifying the control token paths by a compiler is presented and used to find the information that is required for scheduling the program. The analysis is used to find the variables and operations that have an impact on scheduling while the other variables and operations are removed from the model. By removing any variable not related to scheduling, the state space of the program is reduced to make the state space analysis feasible. The resulting model is then used for extracting quasi-static schedules by using the model checking technique presented in [5].

In order to construct a correct scheduling model to be used for actor composition, the control tokens must be modeled such that it is possible to verify that the guards used for a composed actor cover all possible inputs of the composition. To illustrate the approach we will use the, slightly simplified, CAL syntax presented in Table I.

A. Dataflow Analysis (at Instruction Level)

The information we are interested in is the relationship between guards, variables, and ports. We have both global and local variables, $V = V_g \cup V_l$, where global variables mean state variables of the actor and local means variables local to the action that are used to perform the calculation. The actors also have input and output ports, $P = P_i \cup P_o$ and for simplicity, we consider ports to be a special type of variables $P \subseteq V$ and these correspond to local variables according to the patterns specified in the actions (as p:[x] in Table I).

The dependencies between variables in an actor can be described as a binary relation, $R \subseteq V \times V$, which is a subset of the pairs of variables in the actor such that $(x,y) \in R$ indicates that variable x depends on variable y. The relation is built according to the rules in Table III for each actor. For each action of an actor we add to R the variable pairs resulting from the action code and the variables accessed by guards as pairs of guards and variables. We use a special set of variables, $grd_{action} \in V$, to represent the result of evaluating a guard and to indicate that a variable is used in a guard. Table III describes the generation of R from the actors as an annotated type system where a judgement of the form $s : \Sigma \xrightarrow[R]{V} \Sigma$ indicates that statement S modifying the program state between two program states in Σ modifies the variables in V and produces the relations R.

For a single actor, the set of variables used for scheduling can be described as $V_S = \{x \in V \mid (grd, x) \in R^+\}$, where R^+ is the transitive closure of R. For a partition with more than one actor, this is not enough as we also need to take into account the scheduling information passed between actors. To do this, we identify, for each actor, the set of ports used for scheduling $P_g = P \cap V_S$ and use the relation χ from the network description, which describes how ports are connected in the dataflow network, to find the output ports connected to these input ports and add $\{p \in P_o \mid q \in P_i \cap P_g \wedge (q,p) \in \chi\}$ to P_g. These newly added output ports may in turn depend on input ports and P_g must be updated to include $\{q \in P_i \mid (p,q) \in R^+ \wedge p \in P_g\}$ to P_g. We need to iterate these two steps until no new ports are added and finally add the resulting scheduling variables $V'_S = V_S \cup \{x \in V \mid (p,x) \in R^+ \wedge p \in P_g \cap P_o\}$.

The next step is then to transform R into something that gives a simple representation of the variable dependency of the network partition. We simply want to describe the state variables that either a guard or a control output port depends on. The exact behavior is not required and is unnecessarily complex, what is actually needed is the variables and the abstract behavior of these variables. There are only a few types of possible behavior: a guard (or control port) may depend on an input port or on variables, these variables may or may not depend on input ports and may or may not have memory by depending on themselves. We introduce another relation $R_S = \{(x,y) \in R^+ \mid x \in C \vee (x = y \vee y \in P_i) \wedge \exists z \in C, (z,x) \in R^+\}$ where $C = P_g \cap P_o \cup \{grd\}$ to represent the control value dependencies for the current partition, this is a partition specific relation which means that it may not be valid if an actor is added to or removed from the partition.

$$[var]\ x : \{x\}$$

$$[num]\ n : \varnothing$$

$$[op]\ \frac{e_1 : V_1 \quad e_2 : V_2}{e_1\ \text{op}\ e_2 : V_1 \cup V_2}$$

$$[ass]\ \frac{e : V}{x := e : \Sigma \xrightarrow[\{x\}\times V]{\{x\}} \Sigma}$$

$$[skip]\ \ skip : \ \Sigma \xrightarrow[\{\varnothing\}]{\{\varnothing\}} \Sigma$$

$$[seq]\ \frac{S_1 : \Sigma \xrightarrow[R_1]{V_1} \Sigma \qquad S_2 : \Sigma \xrightarrow[R_2]{V_2} \Sigma}{S_1;\ S_2 : \Sigma \xrightarrow[R_1 \cup R_2]{V_1 \cup V_2} \Sigma}$$

$$[if]\ \frac{e : V_c \quad S_1 : \Sigma \xrightarrow[R_1]{V_1} \Sigma \quad S_2 : \Sigma \xrightarrow[R_2]{V_2} \Sigma}{\text{if}\ (e)\ S_1\ \text{else}\ S_2 : \Sigma \xrightarrow[R_1 \cup R_2 \cup ((V_1 \cup V_2) \times V_c)]{V_1 \cup V_2} \Sigma}$$

$$[wh]\ \frac{e : V_c \quad S_1 : \Sigma \xrightarrow[R]{V} \Sigma}{\text{while}\ (e)\ \text{do}\ S_1 : \Sigma \xrightarrow[R \cup (V \times V_c)]{V} \Sigma}$$

$$[action]\ \frac{e : V_g \quad S_1 : \Sigma \xrightarrow[R]{V} \Sigma}{\text{action}\ \ \text{grd}\ e\ \text{do}\ S_1 : \Sigma \xrightarrow[(\{grd\} \times V_g) \cup R]{V_g \cup V} \Sigma}$$

$$[a\_seq]\ \frac{action_1 : \Sigma \xrightarrow[R_1]{V_1} \Sigma \quad action_2 : \Sigma \xrightarrow[R_2]{V_2} \Sigma}{action_1;\ action_2 : \Sigma \xrightarrow[R_1 \cup R_2]{V_1 \cup V_2} \Sigma}$$

$$[actor]\ \frac{action : \Sigma \xrightarrow[R]{V} \Sigma}{\text{actor}\ \ldots\ action \ldots\ \text{end} : R}$$

TABLE III
BUILDING VARIABLE DEPENDENCY RELATION R OF AN ACTOR. THE JUDGMENTS (ARROWS) INDICATES WHICH VARIABLES V ARE ASSIGNED AND WHICH RELATIONS R ARE PRODUCED.

IV. Actor Composition with Control Token Dependency

We use the information generated in the previous section to decide how actors can be composed. The actor compositions of interest is any two actors where one produces a control token which is used by the other actor. The idea is to compose actors with redundant scheduling and for this the scheduler (guards, FSM, priority) of the front actor is used to schedule the composition. The requirement for this to be possible, is that, a specific firing sequence in the front actor implies a specific firing sequence in the second actor. When this is not the case, and the guards does not completely describe the behavior of the composition, we either must transform the front actor to have appropriate guards or not compose the two actors.

The state of an actor is described by the state variables in V_S, and the states of the actor FSM. The scheduling states $\Sigma_S = \{\sigma_1, \sigma_2, ..., \sigma_M\} \subset \Sigma$ is the smallest possible subset of the set of program states, initially including the initial state and the states with input dependent transitions, of the front actor. The generated scheduler is a state machine

$$grd \prec a_1 \prec a_2 \prec a_1 \prec a_2 \prec ..$$
$$\Downarrow$$
$$grd \prec b_1 \prec b_2 \prec b_1 \prec b_2 \prec ..$$

Fig. 1. Action a_2 of the front actor sends an control value that enables a guarded sequence in the second actor, the scheduling of the composed actor is then about interleaving the actions from the two actors after this point.

where the states correspond to the program states Σ_S and the transitions correspond to firing a sequence of actions i.e. a schedule. We define $S_A = \{s_1, s_2, s_3, ..\}$ to be the set of action firing sequences between scheduling states in actor A and $s_i = a_1 \prec a_2 \prec ... \prec a_N$ to be a schedule where a_1 is fired before a_2 etc. For a composition, each state in Σ_S corresponds to some specific value of the variables in V_S and the FSM of the second actor. When a scheduling state is reached with other values on these, we need to add a new state to Σ_S as this state may require slightly different schedules.

A. Control Token Graph and Guard Expressions

We will consider the composition of two actors where the first actor produces a control token consumed by the other. Given two actors, the objective is to find to what extent the scheduling in these is redundant. The two actors can be described as a set of actions such that $A, B \subseteq \{a, b, c, ...\}$. The control tokens produced or consumed are defined as $T \subseteq \mathbb{Z}$. Actor A is the front actor of the composition and it produces a control value for actor B, the objective is to check if firing a control token generating action in A implies that a specific action will become enabled in B (see Figure 1).

We define two relations, $O_A \subseteq A \times T$ and $I_B \subseteq T \times B$ where, O_A is the relation from actions to possible output tokens for actor A and I_B is the function [1] from potential input tokens to actions with a guard accepting this token in actor B. We also define the functions $grd_a(t) \in \{true, false\}$ and $body_a(t) = t'$ representing the guard and calculations in action a where $t, t' \in T$ are the input and output tokens.

We have a relation between the control token production/consumption in the two actors $f : A \times B$ such that $f = \{(a,b) \in A \times B \mid (a,t) \in O_a \wedge (t,b) \in I_b \wedge t \in T\}$. Alternatively we can say that f is the composition of the two relations $f = O_A \circ I_B$. If f is functional, that is, each element in A maps to an unique element in B, then the scheduling of A can be used to schedule B. We can derive the circumstances when there is a functional relation between the schedule in the two actors. To make the discussion easier to follow, the action that produces a control token can be put in one of the groups shown in Figure 2.

We can decide how the control token is produced by actor A, simply by performing some checks on the variable relation R_S and we simply check from which input ports and state variables the control value is generated and whether the state variables in V_S depend on input ports or themselves. The result is the information about if we know the actual values of the control tokens that can be generated or only some properties described by a guard in actor A. For actor B which receives the control token, the control token is either used directly in a guard or passed through one or more variables before ending up in a guard. When the control token passes through variables in actor B, it becomes necessary to prove that we know every possible value of the control token as these variables are used to describe the scheduler state.

[1]We require that I_B is a function, that is, one input value will only enable one action in the specific state. O_A on the other hand is not necessarily a function as an action can produce outputs with different values.

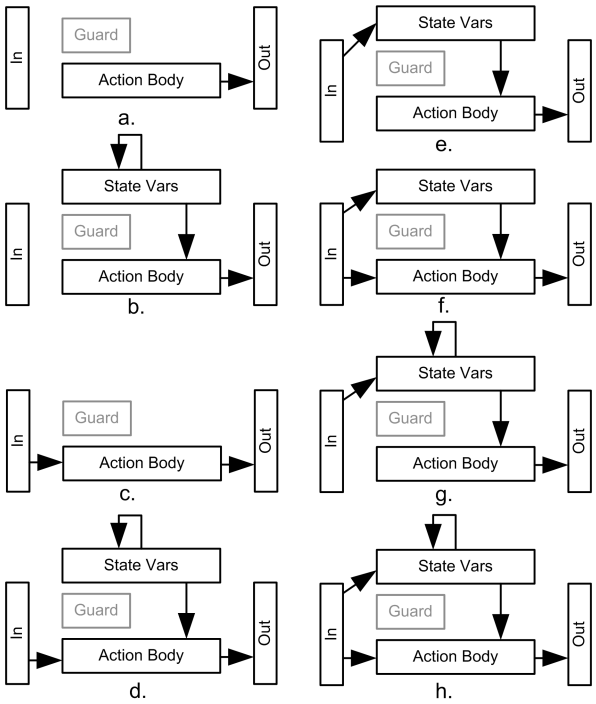

Fig. 2. Possible paths a control token can take through an action: a) generated within the action, i.e. a constant, b) depending on a counter, c) generated from an input value, d) a combination of a counter and an input value, e) generated from previous input values stored in state variables, f) depending on current and previous inputs, and g) depending on a variable with memory of previous inputs.

1) When an Actor is the source of the Control Token: In the first two cases (in Figure 2) the actor is the source of the control token as it does not depend on input in any sense. Case (a) describes the — from a scheduling point of view — best situation, where the action outputs a constant, meaning that the action always produces the same output value. This situation occurs when R_S does not contain a pair connecting the output port to either an input port or a state variable which depend on itself.

$$\forall (x,y) \in R_S : \left[x \in P_g \cap P_o \Rightarrow y \notin P_i \vee (y,y) \notin R_S\right] \quad (1)$$

In the second case (b), there is a variable $(y,y) \in R_S$, this means that the variable updates its value as a function of itself and typically is a counter.

In both cases the control token t' is generated from variables which does not depend on the input t. As there is no input

dependency, each of the variables, related to generating this control value, is seen as having a known value in that state $\sigma_s \in \Sigma_S$ and therefore the output is generated from a set of constants. This means that $O_A = \{(a,t') \mid grd_a(t) = true \wedge t' = body_a(\sigma_s)\}$, and O_A is clearly a function for the state σ_s. As I_b is required to be a function, $f = O_A \circ I_B$ is also a function and A makes the scheduling of B redundant. The actor A is in both cases the actual source of the control value and the generated control values can be determined from the sequence of actions fired. For case b, where there is a counting variable, the scheduler may get more states if there is a scheduling state in actor A which can be reached with different values of the counting variable.

For the second actor, receiving the control token, we know that we cover every possible control token, and for this reason there are no restriction on how this token is used (state variables etc.). Having a counter variable in the receiving actor, of course, may lead to many scheduler states but will not affect the correctness.

2) When Control Token Passes Through Actor: For the following two cases (c and d), the output control value depends on an input port and cannot be derived from the actor only. In case (d) the control token is a combination of the counter and the input value which means that the state of the actor also affects the output value. For a specific state σ_s, including the counter variable, the generated output value is a function of the input and a specific input value always corresponds to a specific output value for a specific state σ_s. This situation is found if R_S connects the control output port to an input port but not through a state variable.

$$\begin{aligned}\exists (p,q) \in R_S : p \in P_g \cap P_o \wedge q \in P_i \wedge \\ \neg \exists x \in V_S : (p,x) \in R_S \wedge (x,q) \in R_S\end{aligned} \tag{2}$$

Here, it is not always the case that the behavior of actor B can be derived from the behavior of actor A. For the composition to make sense, the control token must be used in a guard in actor A before actor B uses it in a guard, this is implied if the action sending the control token have a guard using the input control token (through a peek). The generated control tokens can then be expressed as $O_A = \{(a,t') \in A \times T \mid grd_a(t) = true \wedge t' = body_a(\sigma_s, t)\}$, if $t' = t$ this case also corresponds to two actors sharing the same input token. Now, O_a is not necessarily a function; if grd_a accepts more than one value then O_a may contain several output values for one action. It can still be possible to show that f is functional by showing that Proposition 3 holds.

$$\forall (a,t_1),(a,t_2) \in O_A : \left[(t_1,b),(t_2,c) \in I_B \Rightarrow b = c\right] \tag{3}$$

We also get more restrictions for the second actor. If the second actor has a guard directly reading the control token from the input port (peek) and there is a functional dependency between the guards in the two actors, the actors can be composed. If the second actor reads the control value to a variable before using it in a guard, this variable is also used to describe the scheduler state and may introduce new states in Σ_S. The problem is that the guard often accepts a wide range of values and the program behavior should be analysed for each of these.

3) When there are Input Dependent Control Variables: The last four cases have one thing in common - the control token is produced from an input port and passes through a state variable.

$$\exists (p,x),(y,q) \in R_S : \left[p,q \in P_g \wedge x = y\right] \tag{4}$$

For a composition using the guards of the front actor to be possible, either the action setting the value of the variable from the input port or the action setting the value of the output port from the variable must have a guards to be compared with the guard in the receiving actor. To resolve these cases, it is necessary to analyze the order in which these variables are read and written and to check at what point there is a guard using this value. An alternative solution is to remove the variable from the scheduler state and instead use its value in the same fashion as an input port. The strategy for scheduling each of these cases is to, if possible, transform the actor to resemble case c and then perform the analysis accordingly.

Case e may correspond to case c if the action a_1 that sets the variable from the port precedes the action a_2 that writes the control token, $\forall s \in S_A : b \in s \Rightarrow a \in s \wedge a \prec b$, if these actions are always fired in sequence, they could be merged and correspond to case c. If this is not the case, the state variable must be seen as an input port and the schedule with the action producing the control token must have a guard depending on this variable. In this case, the same rules as for case c applies.

In the sixth case (f) the output is a combination of current and previous inputs. As we cannot assume that we know every possible input that can be read to the state variable, the state variable must be seen as an input port in the same fashion as case e, then for the guard using this variable, we can apply the same rules as for case c. The same applies for cases g and h, where the state variable depends on input but also on the history of (potentially all) previous inputs. The variable is likely to have many states and should not be part of the scheduler state, instead we can consider the variable to act as an input port, but only in the case the front actor uses this variable in a guard which either is the same as the one producing the control or precedes it in each schedule.

B. Partitioning, Composition, and Scheduling

From the discussion above it is clear that the scheduling of some compositions with control tokens requires much less work from the compiler or design tools than other. What can easily be identified are the actors which are the source of the control token. When, for some reason, a partitioning where the front actor is not the source of the control token is requested, it may still be possible to show that there is a functional relation between the actions in the actors making the composition feasible. Otherwise, searching *upstream* for the actor with the source and performing that composition first, may resolve the problem. Figure 3 describes the different

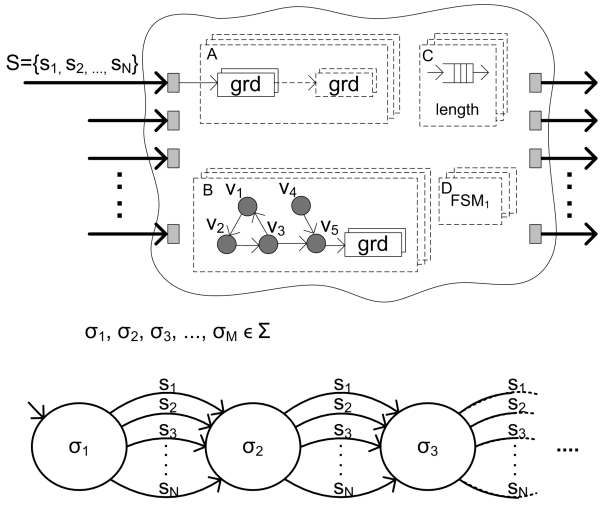

Fig. 3. The state of the partitions is defined by state variables, FSMs and FIFO states. The scenarios of the partitons are decided by the input sequences S. The number of schedules needed is $M \times N$ where M is the number of scheduling states and N is the number of input sequences.

aspects of a partition constructed for composition, here A describes the control tokens entering a partition and how these may be propagated inside the partition. For the scheduling of a partition, it is these guards that need to be analyzed as described above.

While there are restrictions on control values entering a partition from the outside, control tokens with dependencies inside the partition only, can not affect the correctness of the model and therefore we *can* allow any type of variable dependency inside the partition. This kind of control structure is illustrated as B in Figure 3 where a guard depends on several variables, possibly from different actors, but does not depend on the partition inputs. Depending on the structure of the variable dependency, some variables, where there is a cyclic dependency, can hold many values and cause the partition scheduler to have many states.

A scheduler for a partiton can be described as the FSM in Figure 3 where each state has an outgoing transiton corresponding to the possible input sequences of the partition. How the transitions connect the states depend on what the program state is after the corresponding schedule and does not necessarily follow the example in Figure 3. The number of states needed in the FSM depends on the number of states the rest of the partition will end up in when the actor used as front actor executes between its scheduling states. The number of schedules needed is the number of states times the number of alternative transitions in the scheduling states.

V. Case Study and Experimental Results

The presented analysis is implemented as a set of compiler passes generating graphs describing scheduling dependencies. The idea is that the compiler can use this information to decide how to partition the program and whether to compose some actors into a larger actor or run these separately; also the programmer could use this information to find and remove

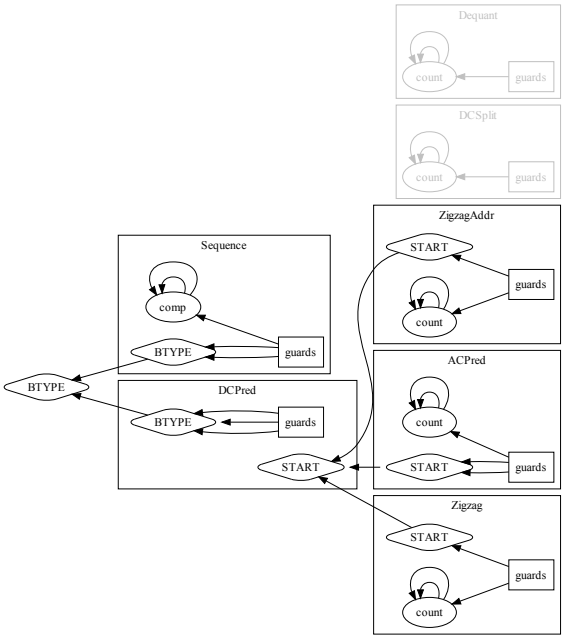

Fig. 5. An abstract dependency graph for the intra prediction network of an MPEG-4 decoder. Circles represent state variables, diamonds represent ports.

Scenario	Port	read/write
start	BTYPE	1 (2048) + 2 (data)
	START	1 (constant)
inter_ac	BTYPE	1 (514)
	START	1 (constant)
other	BTYPE	1 (0)
	START	1 (constant)
intra	BTYPE	1 (1024)
	START	1 (constant in if-statement)

TABLE IV
The different scenarions of the *DCPred* actor. The table shows the action starting a scenario, the control port affected and the pattern of this port in the specific scenario.

unnecessary dependencies from the program or simply to create partitions manually.

To demonstrate the approach we will use the texture processing part of an MPEG-4 decoder as example. The control value graph shown in Figure 5 gives a graphical view of the relation R_S, for the decoder sub-network, where ovals and diamonds represent variables and ports respectively and arrows indicate that there is a dependency in R_S.

The graph of this sub-network shows three different types of dependencies between actors, we will investigate how the actor *DCPred*, which according to the graph has control dependencies to several of the other actors of the network, can be used to schedule the partition. This actor depends on an input value from outside the partition, shares this same input value with one of the actors (*Sequence*) and produces the control tokens for three of the actors. Two actors do not depend on any values from other actors.

The first dependency to be resolved is the one between the two actors sharing the control value from outside the partition and to find the guards that describes the behavior of these.

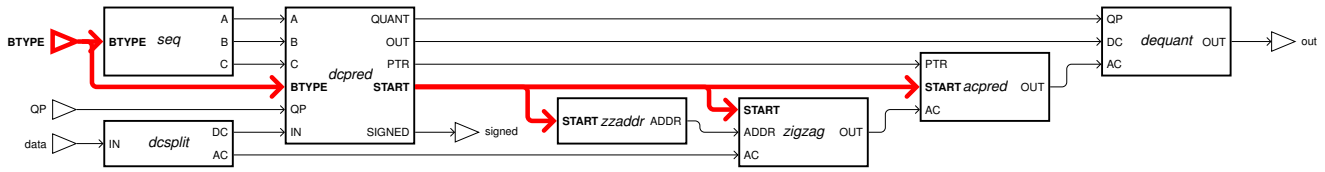

Fig. 4. The intra prediction network of a RVC-CAL MPEG-4 decoder, with the queues carrying control tokens high-lighted (red color).

In practice this means that we must show that the guards of these actors are compatible; according to Proposition 3 in Section IV-A there must be, for each token accepted by one of the guards in actor A, one specific guard in actor B that accepts these tokens. The idea is to use a similar technique as in [4] to check this property, in this case study it was done by hand to give an example of the property. Figure 6 describes the resulting relation, it shows that the relation is functional as for each accepted input in *DCPred*, there is a specific guard in *Sequence* that accepts this token. This means that we can use the guards of *DCPred* to schedule *Sequence* but note that the opposite would not work.

The second dependency to be resolved is the control value sent from *DCPred* to three of the other actors in the sub-network. The graph in Figure 5 shows that the control value is produced from constants and does not depend on input nor on the state of the actor, Table IV, however, shows that the *intra* scenario chooses one of several possible constants inside an if-statement. In the case of an if-statement, the options are to either keep a dependency to the condition and include it in the model or remove this dependency and check how the different constants affect the scheduling. In this case, as this if-statement depends on input values, we remove it and analyze the different possible cases.

We know that the output value is generated from constants and that there will be one value from each case in the if-statement, the values of the outputs can therefore be found, simply by executing each of the branches. Figure 7 shows the relation between the scenarios in *DCPred*, the output values, and the scenarios in *ACPred*; the relation to the other two actors using this control value is omitted as these are similar to this one. According to Section IV-A the relation between the guards of the scenarios must be functional for the scheduling of the second actor to be redundant. For Figure 7, we can see that while the relation from *DCPred* to the output value is not functional, the relation from *DCPred* to *ACPred* is functional as the guard of *start* in *ACPred* accepts each of the values generated by *intra* in *DCPred*.

The result of the analysis of this specific network, is, that the scheduler (guards, FSM) of the *DCPred* actor completely describes how to schedule the rest of this partition. The actors of this partition can therefore be composed into one single actor, the benefits of this is that every scheduling decision of the other actors than *DCPred* will be removed and the FIFOs inside the composition can be replaced with variables/arrays as soon as the action firing sequences have been specified by methods such as [5].

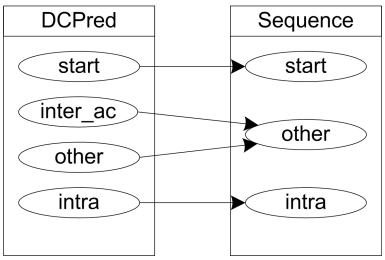

Fig. 6. The relation between the guards of actions triggering each of the scenarios in the two actors sharing the control input value of the sub-network.

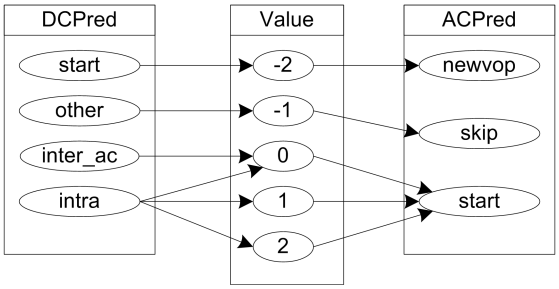

Fig. 7. The relations between the scenario run in *DCPred*, the produced control value and the guards accepting these control values in *ACPred*.

A. Resulting Scheduler

The scheduler for the composed network partition can be based on the *DCPred* actor. This actor has one FSM state where the actions depend on input and these actions describes the different types of input the network can process. From this we know that the scheduler is an FSM with at least one state and four transitions, corresponding to schedules leaving that state. If it is not possible to construct each schedule such that it consumes the inputs and runs the partition to a state where the FSMs and scheduling state variables V_S corresponds to the initial state, more states are needed to describe the scheduling of the partition.

For the actual scheduling, the variables in Figure 5 and the FSM states of the actors define the state of the partition. The internal FIFOs of the partition are required to be empty after a schedule has terminated. Using the model-checking technique in [5], we generate a model-checker using the Promela backend of the Orcc compiler and provide input accepted by each of the four guards according to Table IV. The actual schedules are generated by searching for a path to a state where this input has been consumed and the *DCPred* actor is ready for the next control input. Then, we check if the

Guard	Nr. actions (State0)	Nr. actions (State1)
start	9	268
inter_ac	135	330
other	8	267
intra	136	331

TABLE V
THE LENGTH OF THE SCHEDULES GENERATED FOR THE EXAMPLE NETWORK. A SCHEDULE IS CHOSEN BASED ON WHICH GUARD EVALUATES TO TRUE AND THE CURRENT SCHEDULER STATE. THE NUMBERS INDICATE HOW MANY ACTIONS ARE FIRED BASED ON ONE GUARD EVALUATION.

found state is an already known state or if we must add this state to the scheduler. When a new state is required, schedules for each input type are also generated for this state. When each state has, in this case, four transitions connected to a known state, the scheduling is completed.

The scheduling of this network results in a scheduler with two states and eight transitions representing static schedules (see Table V). The reason for the second state is that in this particular implementation, the *zigzag* fills an internal buffer at the first block of a frame and flushes the buffer at a new frame, having different FSM states indicating whether the buffer is filled or not. This implies that two different schedules are needed for each block type. The scheduling variables named *count* each return to the initial value after each schedule, the scheduling variable *comp*, however, does not as it indicates which block of the macro block currently is processed. For this reason this scheduling decision needs to be taken at runtime, either by adding more states to the scheduler or, as in this case was possible by first applying the tools presented in [4], merging the actions depending on this variable and removing the variable from the scheduling state space.

A scheduler for the partition in this case study, can be described as in Table VI. This scheduler is somewhat simplified and indicates that we need to check that the control token is available and based on the guards from *DCPred*, choose the appropriate schedule. The schedules are sequences of actions that can fire without any further guard evaluations. Comparing this result to [5], we end up with a corresponding set of schedules for this network, and obviously similar results regarding speed-up; about 22% increase in frame rate for the texture coding part (acdc, idct) of the decoder. However, the scheduler is generated with evidence that the set of schedules completely describes the original program behavior and works for all possible input while in [5] this was verified manually by inspecting the code.

```
void decoder_acdc_scheduler() {

  if ( has_tokens(BTYPE, 1) )
    btype = peek(BTYPE)
  else return

  if (state == 0)
  {
    if (btype == grd_DCPred_start)
      dcPred_start();
      acPred_newvop();
      ...
      state = 0;
    else if (btype == grd_DCPred_inter_ac)
      dcPred_inter_ac();
      acPred_start();
      ...
      state = 1;
    else if ...
  }
  else if (state == 1)
  {
  ...
  }
}
```

TABLE VI
EXAMPLE PSEUDO CODE OF GENERATED SCHEDULER.

VI. CONCLUSIONS

In this paper we have shown how control tokens can be modelled to enable composition and scheduling of dynamic dataflow programs. Control token paths are analyzed in an abstract manner to deduce if there is a real data dependency or if a control token is used to distribute some information in the dataflow network, which, if the actors are composed, can be removed. As a result, many of the guards can be resolved at compile-time, resulting in less scheduling overhead at runtime. The actual speed-up a program gains, however, depends on how large partitions are composed and how well these fit on the target platform i.e. how well the schedules fit in the cache. The goal with the methods presented in this paper is to make the control tokens of a dataflow program a visible part of the model such that actors that can be efficiently composed are highlighted and can be scheduled using existing methods.

REFERENCES

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

[2] M. Mattavelli, I. Amer, and M. Raulet, "The reconfigurable video coding standard," *IEEE Signal Processing Magazine*, vol. 27, no. 3, pp. 157–167, 2010.

[3] J. Boutellier, M. Raulet, and O. Silvn, "Automatic hierarchical discovery of quasi-static schedules of rvc-cal dataflow programs," *Journal of Signal Processing Systems*, vol. 71, no. 1, pp. 35–40, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11265-012-0676-4

[4] M. Wipliez and M. Raulet, "Classification of dataflow actors with satisfiability and abstract interpretation," *IJERTCS*, vol. 3, no. 1, pp. 49–69, 2012.

[5] J. Ersfolk, G. Roquier, F. Jokhio, J. Lilius, and M. Mattavelli, "Scheduling of dynamic dataflow programs with model checking," in *IEEE International Workshop on Signal Processing Systems (SiPS)*, 2011.

[6] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static data flow," in *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, vol. 5, May, pp. 3255–3258 vol.5.

[7] B. Bhattacharya and S. Bhattacharyya, "Parameterized dataflow modeling of dsp systems," in *Acoustics, Speech, and Signal Processing, 2000. ICASSP '00. Proceedings. 2000 IEEE International Conference on*, vol. 6, pp. 3362–3365 vol.6.

[8] R. Gu, J. Janneck, M. Raulet, and S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," *Journal of Signal Processing Systems*, vol. 63, pp. 129–142, 2011.

[9] J. Janneck, "A machine model for dataflow actors and its applications," in *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, nov. 2011, pp. 756 –760.

[10] J. Ersfolk, G. Roquier, J. Lilius, and M. Mattavelli, "Scheduling of dynamic dataflow programs based on state space analysis," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, march 2012, pp. 1661 –1664.