Classification of General Data Flow Actors into Known Models of Computation

Christian Zebelein, Joachim Falk, Christian Haubelt, and Jürgen Teich
Hardware/Software Co-Design
University Erlangen-Nuremberg, Erlangen, Germany
E-mail: `{christian.zebelein,haubelt,falk,teich}@cs.fau.de`

Abstract

Applications in the signal processing domain are often modeled by data flow graphs which contain both dynamic and static data flow actors due to heterogeneous complexity requirements. Thus, the adopted notation to model the actors must be expressive enough to accommodate dynamic data flow actors. On the other hand, treating static data flow actors like dynamic ones hinders design tools in applying domain-specific optimization methods to static parts of the model, e.g., static scheduling. In this paper, we present a general notation and a methodology to classify an actor expressed by means of this notation into the synchronous and cyclo-static data flow models of computation. This enables the use of a unified descriptive language to express the behavior of actors while still retaining the advantage to apply domain-specific optimization methods to parts of the system. In experiments we could improve both latency and throughput of a general data flow graph application using our proposed automatic classification in combination with a static single-processor scheduling approach by 57%.

1. Introduction

Typical signal processing algorithms as can be found in multimedia applications are often modeled by data flow graphs. Such data flow models often permit the use of efficient domain-specific optimization methods during the design process. As an example, one might think of computing static and buffer minimal schedules for synchronous data flow graphs [13] (*SDF*), i.e., data flow graphs with constant consumption and production rates. However, due to the complexity of today's multimedia applications, these algorithms can no longer be modeled using static data flow models only. They are rather modeled by instances of static, dynamic, or even non-deterministic data flow actors connected in a single data flow graph.

As a consequence, design methods applicable to static data flow graphs cannot be used in the presence of such highly expressive models. On the other hand, applying domain-specific methods to appropriate subgraphs still might result in improved implementations. In order to use this potential, these appropriate subgraphs must be identified inside a general data flow model. As an example, consider the Motion-*JPEG* decoder shown in Figure 1 modeled as a Kahn Process Network (*KPN*) [11]. It exhibits actor complexities varying from static (the shaded actors in Figure 1) to fully dynamic (white). The class of non-deterministic actors is missing from this example. However, our proposed formalism can express also this actor class without difficulties.

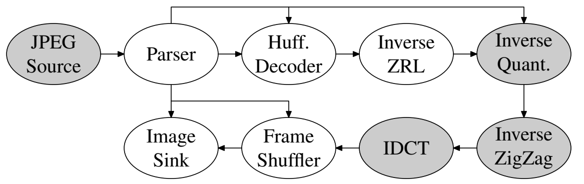


Figure 1. Motion-*JPEG* decoder consisting of dynamic and static data flow actors (shaded).

In this paper, we will present a formal model in which each actor is described by a finite state machine controlling the communication behavior of the actor and atomic actions performing data transformations. This model permits the classification of actors, and hence the identification of synchronous data flow (*SDF*) and cyclo-static data flow (*CSDF*) [5] subgraphs. As the problem of actor classification in its general form is undecidable, we will discuss the limitations of our proposed approach.

To make this new methodology useful in real world design flows, we show how to apply our algorithm to data flow models specified with SystemC [8, 1], a design language mainly used in the early prototyping phase for functional simulation. In our experiments, we could improve latency and throughput of the Motion-*JPEG* decoder from Figure 1 designed with SystemC using our proposed classification algorithm combined with static scheduling by 57%.

The remainder of this paper is organized as follows: Section 2 formally defines our model and its relation to data flow applications based on SystemC. Subsequently, Section 3 presents the actor classification algorithm, while Section 4 will discuss the recognized subset of actors identifiable as *SDF* and *CSDF*. Related work will be discussed in Section 5 and experimental results are presented in Section 6.

2. General Data Flow Model

In this section, we present our abstraction for SystemC applications, e.g., the Motion-*JPEG* decoder presented before. At a glance, the notation extends conventional data flow graphs by finite state machines controlling the consumption and production of tokens by actors. As known from data flow descriptions the topology of the actor connections is modeled by a directed graph in the following called *network graph*. An actor a can only communicate with other actors through its sets of *input* and *output ports*, denoted $a.I$ and $a.O$, respectively. These ports are connected with each other via *FIFO* channels. The basic entity of data transfer are *tokens* transmitted via these channels. In Figure 1 an example for such a data flow graph is depicted, with edges and vertices representing *FIFO* channels and *actors*, respectively. However, due to the missing relevance of the network graph for the actor classification we will abstain from a formal definition. The classification algorithm is based on the information provided by the formal model of an actor. In particular, it depends on the finite state machine of a given actor which controls the consumption and production of tokens. A visual representation of all parts of the formal actor description is given in Figure 2.

As can be seen from Figure 2 an actor a is composed of three major parts: (i) actor input and output ports $a.I$ and $a.O$ which are simply required to select source or destination FIFO channels for a given read or write operation, (ii) some form of functionality $a.\mathcal{F}$, e.g., f_1 or g_1, triggered by (iii) the previously mentioned finite state machine $a.\mathcal{R}$, the so-called actor *FSM*. More formally, an actor can be defined as follows:

Definition 1 (Actor) *An* actor a *is a tuple* $(I, O, \mathcal{F}, \mathcal{R})$ *containing a set of* actor input ports I*, a set of* actor output ports O*, the* actor functionality $\mathcal{F}$ *and the* finite state machine *(*FSM*)* $\mathcal{R}$*.*

The *FSM* and the *functionality* represent a separation of the control flow governing the consumption and production of tokens by the actor and the data flow path in the actor, i.e., computation on token values. More formally, we can derive the following definition for the *actor functionality*:

Definition 2 (Actor functionality) *The* actor functionality $a.\mathcal{F}$ *is a tuple* $(F_{action}, F_{guard}, S_{func}, s_{0,func})$ *containing a set of* functions F *partitioned into* actions F_{action} *and* guards F_{guard}*, a set of* functionality states S_{func} *(possibly infinite), and an* initial functionality state $s_{0,func} \in S_{func}$.

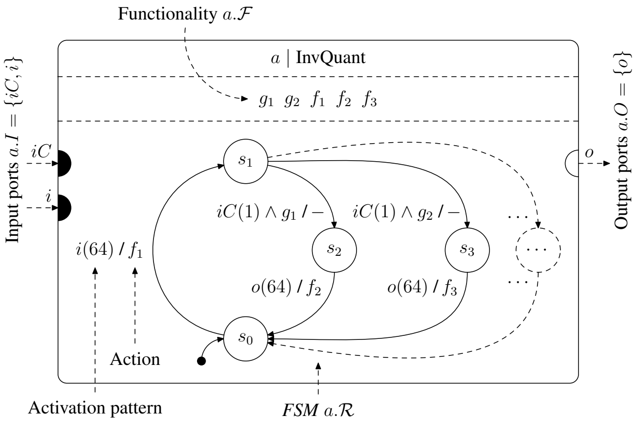


Figure 2. A visual representation of the formal model of the `InvQuant` actor.

Actions and guards differ in their ranges. A guard function evaluates to a boolean value, i.e., $f_{\text{guard}} : V^n \times S_{\text{func}} \rightarrow \{\text{true}, \text{false}\}$, and its evaluation to $\mathtt{true}$ is a prerequisite to enable the transition annotated with this guard. In contrast to this an action is triggered by the execution of a transition and evaluates to a sequence of tokens $\mathbf{v} \in V^m$, which are produced on selected actor output ports, and a successor functionality state, i.e., $f_{\text{action}} : V^n \times S_{\text{func}} \rightarrow V^m \times S_{\text{func}}$. The input token sequence for actions and guard functions, i.e., $\mathbf{v} \in V^n$ is provided by the actor *FSM* which retrieves those values from the tokens in the *FIFO* channels connected to the actor input ports.

The execution of an actor is divided into atomic *firing steps*, each of which consists of three phases: (i) checking for enabled transitions of each actor, (ii) selecting and executing one enabled transition per actor, and (iii) consuming and producing tokens needed by the transition. More formally, an actor FSM is defined as follows:

Definition 3 (Actor FSM) *The* actor FSM $a.\mathcal{R}$ *is a tuple* $(T, S_{fsm}, s_{0,fsm})$ *containing a set of* transitions T*, a set of* states S_{fsm} *and an* initial state $s_{0,fsm} \in S_{fsm}$*. A* transition t *is a tuple* $(s_{fsm}, k, f_{action}, s'_{fsm})$ *containing the current state* $s_{fsm} \in \mathcal{R}.S_{fsm}$*, an* activation pattern k*, the associated* action $f_{action} \in \mathcal{F}.F_{action}$*, and the next state* $s'_{fsm} \in \mathcal{R}.S_{fsm}$*. An* activation pattern k *is a tuple* (R, G)*. The function* $R : a.I \cup a.O \rightarrow \mathbb{N}_0$ *specifies for each actor input port* $p \in a.I$ *the number of tokens consumed and for each output port* $p \in a.O$ *the number of tokens produced. The set of guards* $G \subseteq \mathcal{F}.F_{guard}$ *represents more general predicates depending on the current functionality state or the token values on the input ports.*

As can be seen from the previous definitions both actor *FSM* and functionality contain a state space. Therefore, the whole state space of an actor must be contained in the cross product of both state spaces. More formally, we define an *actor state* as follows:

```
SC_MODULE(InvQuant) {
  ...
  sc_fifo_in <int>               iC; // in port
  sc_fifo_in <FreqQuantized_t> i;  // in port
  sc_fifo_out<FreqCoeff_t>       o;  // out port
  ...
  void process() {
    FreqQuantized_t inCoeffs[64];

    while (true) {
      // need fixed iteration count
      for (int n = 0; n < 64; ++n) // code f_1
        inCoeffs[n] = i.read();
      int quantTable = iC.read();
      // select quantization table
      switch (quantTable) {
      case 0:
        // need fixed iteration count
        for (int n = 0; n < 64; ++n) // code f_2
          o.write(inCoeffs[n]*quantTable0[n]);
        break;
      case 1:
        // need fixed iteration count
        for (int n = 0; n < 64; ++n) // code f_3
          o.write(inCoeffs[n]*quantTable1[n]);
        break;
      ...
      }
    }
  }
};
```

Figure 3. SystemC pseudo code implementing the `InvQuant` **actor.**

Definition 4 (Actor state) *The* current actor state *of an actor a is the pair $s = (s_{func}, s_{fsm})$ consisting of the current functionality state $s_{func} \in \mathcal{F}.S_{func}$ and the current state $s_{fsm} \in \mathcal{R}.S_{fsm}$ of the actor* FSM. *The* actor state space *$S \subseteq \mathcal{F}.S_{func} \times \mathcal{R}.S_{fsm}$ (possibly infinite) contains all actor states reachable during the execution of a, with the initial actor state being defined as $s_0 = (\mathcal{F}.s_{0,func}, \mathcal{R}.s_{0,fsm})$.*

Next, we will discuss how to derive our formal notation from the SystemC application. For this purpose, we require the input SystemC application to only communicate via SystemC *FIFO*s and that its functionality is implemented via a single SystemC thread (`SC_THREAD`). As can be seen from the code of the `Inverse Quantization` actor in Figure 3 the actor input and output ports from the notation level have a natural correspondence at the source code level, whereas the actor FSM and functionality are intermingled in the `process` method implementing the SystemC thread.

We only sketch the idea to derive both aspects from the SystemC `process` method. In effect, we start with the *control data flow graph* (*CDFG*) of the method, as derived by any C++ parser, and collapse all strongly connected subgraphs of the CDFG not containing any read or write *CDFG* nodes into a single opaque node. Subsequently, we do loop unrolling for loops with known boundaries. Note that these are only loops with loop bodies containing read or write operations, due to the collapse of all other loops into opaque nodes. Finally, we collect sequences of read or write operations on a given port into a single node and transition annotated with the number of tokens consumed and produced. An example of the *FSM* derived by this approach from the code of the `process` method is shown in Figure 2.

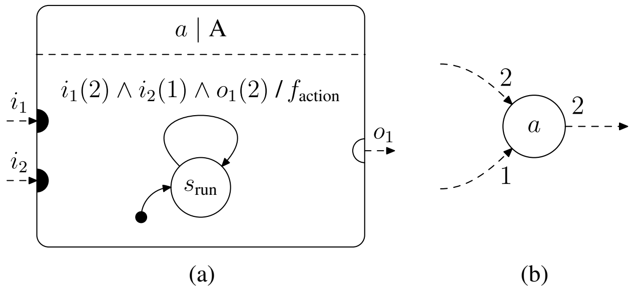


Figure 4. (a) The possible state transition of actor a consumes two tokens from i_1, one token from i_2, and produces two tokens on o_1. (b) Equivalent *SDF* actor.

3. Classification

In order to be able to apply domain-specific design methods, we need to recognize important data flow models of computation such as *SDF* and *CSDF*. We will show that this can be accomplished by the analysis of the *FSM* of an actor in our proposed model.

The most basic representation of an *SDF* actor in our general data flow notation can be seen in Figure 4(a): The *FSM* $a.\mathcal{R}$ contains only one transition which consumes two tokens from the input port i_1, one token from the input port i_2, and produces two tokens on the output port o_1. Clearly, the actor exhibits a static communication behavior, which makes it possible to visualize the actor as commonly known from *SDF* graphs, depicted in Figure 4(b).

It may be noted that in *SDF* and *CSDF* graphs the actors are connected via *infinite* channels, i.e., an actor can fire without having to check if enough free space is available on its output ports. In our model, however, the *FIFO* channels are finite. As a result, fully dynamic scheduled actors must also specify the number of tokens produced by each transition so that the scheduler selects only valid transitions for execution. On the other hand, we assume that for subgraphs classified into the *SDF* or *CSDF* model of computation, a static schedule will be calculated. In this case, an upper bound for the size of each *FIFO* in the subgraph can be determined so that the static schedule can be executed without having to check for free space on output ports.

The idea of the proposed classification algorithm is to check if the communication behavior of a given actor can be reduced to a basic representation which can be easily classified into the *SDF* or *CSDF* model of computation. The

FSM describes the *possible* communication behavior of the actor. However, due to the presence of guards which depend on the functionality state of the actor, only a subset of transitions may be enabled during the execution of the actor. This refinement of the possible communication behavior will be captured by the introduction of the so-called *dynamic state transition diagram*, which contains the actor state space from Definition 4.

Based on the dynamic state transition diagram, some requirements are then identified which must be met by both the actor and the classification algorithm in order to avoid the introduction of deadlocks into the model by spuriously treating the actor as an *SDF* or *CSDF* actor. Subsequently, the actual classification algorithm will be presented.

The dynamic state transition diagram may be infinite, hence being intractable by the algorithm. Moreover, no tools based on our model exist for the automatic construction of this diagram. Therefore, we will present a *reduced state transition diagram* based on the actor *FSM*, which does not consider the functionality state of the actor. This simplification restricts the set of actors with a static communication behavior which will be recognized by the algorithm. This will be discussed in Section 4.

To simplify the notation, we assume in the following that always the same actor a is considered, so that the members of a as introduced in Definition 1 can be accessed without using the '.'-operator. Additionally, we need to enumerate the input and output ports of a, which is why we assume that they are given as ordered sets $I = \{i_1, i_2, \ldots, i_n\}$ and $O = \{o_1, o_2, \ldots, o_m\}$, respectively.

To accommodate *CSDF* actors, we use $\tau(a)$ to refer to the number of phases. To avoid complicated case differentiations, *SDF* actors will simply be treated as *CSDF* actors with $\tau(a) = 1$.

In order to be able to apply some vector operations to the (assumed) consumption and production rates of the actor, we define a function $\mathbf{cns} : \mathbb{N} \to \mathbb{N}_0^{|I|}$ which maps a specific actor phase ρ, $1 \leq \rho \leq \tau(a)$ to a vector of the dimension $|I|$ whose pth component ($1 \leq p \leq |I|$) contains the number of tokens consumed from the input port $i_p \in I$ by the activation of a in phase ρ. Analogously, $\mathbf{prd} : \mathbb{N} \to \mathbb{N}_0^{|O|}$ maps a specific actor phase ρ, $1 \leq \rho \leq \tau(a)$ to a vector of the dimension $|O|$ whose pth component ($1 \leq p \leq |O|$) contains the number of tokens produced on the output port $o_p \in O$ by the activation of a in phase ρ.

Dynamic state transition diagram The communication behavior of an actor during its execution can be described by means of its actor state space. To refer to the set of *all* outgoing transitions of a specific state, we define a function $\delta : \mathcal{R}.S_{\text{fsm}} \to 2^{\mathcal{R}.T}$, such that $\delta(s_{\text{fsm}}) = \{\, t \in \mathcal{R}.T \mid t.s_{\text{fsm}} = s_{\text{fsm}} \,\}$. To refer to the set of *enabled* outgoing transitions of a specific state, we define a function $\epsilon : \mathcal{F}.S_{\text{func}} \times \mathcal{R}.S_{\text{fsm}} \to 2^{\mathcal{R}.T}$, such that $\epsilon(s_{\text{func}}, s_{\text{fsm}}) = \{\, t \in \delta(s_{\text{fsm}}) \mid t \text{ is enabled under the functionality state } s_{\text{func}} \,\}$. An actor a is *blocked* in $s \in S$ if $\epsilon(s) = \emptyset$. Otherwise, a transition is non-deterministically chosen from $\epsilon(s)$ and executed. The actor state space from Definition 4 together with the set of enabled outgoing transitions of a specific state constitute the so-called *dynamic state transition diagram*:

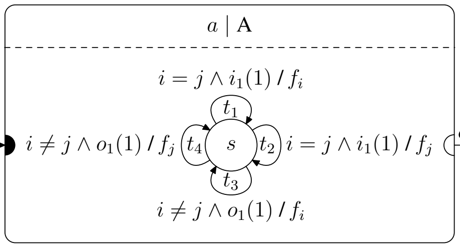
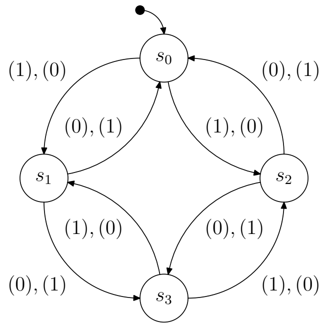

$a \mid A$
$i = j \wedge i_1(1)\,/\,f_i$
t_1
i_1
$i \neq j \wedge o_1(1)\,/\,f_j$
t_4
s
t_2
$i = j \wedge i_1(1)\,/\,f_j$
o_1
t_3
$i \neq j \wedge o_1(1)\,/\,f_i$

Figure 5. The *FSM* of actor a consists of the state s and four transitions $t_1, \ldots, t_4$. The set of functionality states is $\mathcal{F}.S_{\text{func}} = \{\, (i,j) \mid i,j \in \mathbb{N}_0 \,\}$, with the initial functionality state $\mathcal{F}.s_{0,\text{func}} = (0,0)$. The action f_i is defined as $f_i : i \leftarrow (i+1)\,mod\,2$, whereas f_j is defined as $f_j : j \leftarrow (j+1)\,mod\,2$.

Definition 5 (Dynamic state transition diagram) *The* dynamic state transition diagram *of an actor a is the tuple $G_d = (S, E, s_0)$ containing the actor state space S and the initial actor state $s_0 \in S$ (see Definition 4). The edges $E \subseteq S \times \mathbb{N}_0^{|I|} \times \mathbb{N}_0^{|O|} \times S$ consist of tuples $e = (s, \mathbf{cns}, \mathbf{prd}, s')$. For each actor state $s \in S$ and each enabled outgoing transition $t \in \epsilon(s)$ of s, an edge e is added to the set of edges E connecting the current actor state $e.s = s$ to the next actor state $e.s' = (s'_{func}, t.s'_{fsm})$, with s'_{func} being the new functionality state after the execution of $t.f_{action}$. The component $e.\mathbf{cns}$ is a vector of the dimension $|I|$ whose pth component ($1 \leq p \leq |I|$) contains the number of tokens consumed from the input port $i_p \in I$ during the execution of t, i.e., $e.\mathbf{cns} = (t.k.R(i_1), \ldots, t.k.R(i_{|I|}))$. Analogously, $e.\mathbf{prd}$ is a vector of the dimension $|O|$ whose pth component ($1 \leq p \leq |O|$) contains the number of tokens produced on the output port $o_p \in O$ during the execution of t, i.e., $e.\mathbf{prd} = (t.k.R(o_1), \ldots, t.k.R(o_{|O|}))$.*

Figure 5 shows an actor whose dynamic state transition diagram G_d is to be constructed. The actor state space consists of only four states, depicted in Figure 6. The edges of G_d are constructed as described in the above definition, although they are only annotated with the vectors $e.\mathbf{cns}$ and $e.\mathbf{prd}$. If the actor is currently in the initial state s_0, both transitions t_1 and t_2 are enabled (because $i = j = 0$ initially), assuming at least one token is available on the input port i_1. As a result, two edges will be added to G_d: the first edge connects s_0 to the state s_1 (corresponding to the execution of t_1), while the second edge connects s_0 to the state



(a) G_d of actor a from Figure 5.

$s \in S$	$(s_{\text{func}}, s_{\text{fsm}})$
s_0	$((0,0), s)$
s_1	$((1,0), s)$
s_2	$((0,1), s)$
s_3	$((1,1), s)$

(b) The actor state space.

Figure 6. The actor state space evaluates to $S = \{s_0, s_1, s_2, s_3\}$, with the initial state s_0 (b).

s_2 (corresponding to the execution of t_2). Both transitions consume one token from the input port i_1, which results in $e.\mathbf{cns} = (1)$ and $e.\mathbf{prd} = (0)$ for both edges.

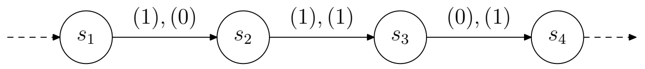
It may be noted that G_d is infinite if the actor state space S is infinite. However, the classification algorithm has to visit all outgoing edges of all states, and won't terminate in such a case. Therefore, the classification algorithm cannot be applied to such an infinite dynamic state transition diagram. To allow classification also for those actors, one can use a *reduced state transition diagram*, presented later, which does not depend on the functionality state of the actor.

Classification requirements Independent from the actually supplied state transition diagram for the algorithm, the following properties must hold true for the dynamic state transition diagram of an actor in order to avoid the introduction of deadlocks into the transformed model that treats an actor as an *SDF* or *CSDF* actor:

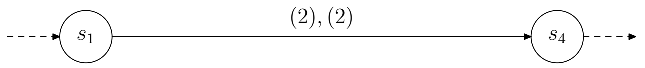
Definition 6 (Liveliness) *Each actor state $s \in S$ must have at least one enabled outgoing transition, i.e., $\forall s \in S : \epsilon(s) \neq \emptyset$. This is required to be able to activate the equivalent* SDF *or* CSDF *actor an infinite number of times: If an actor state $s \in S$ with $\epsilon(s) = \emptyset$ existed, the actor would be blocked in this state forever, possibly resulting in a deadlock of the model.*

In the following, a *path of edges* $(e_1, e_2, \ldots)$ in a state transition diagram is a (possibly infinite) sequence of consecutive edges, i.e., $\forall i \geq 1 : e_i.s' = e_{i+1}.s$. A *cyclic path* is therefore a path $(e_1, e_2, \ldots, e_n)$ with $e_n.s' = e_1.s$. We will use the '$\frown$'-operator to denote the concatenation of paths.

Definition 7 (Active input–output behavior) *The actor has to consume or produce tokens every now and then, i.e., no infinite path of edges $(e_1, e_2, \ldots)$ in the dynamic state transition diagram must exist so that $\forall i \geq 1 : e_i.\mathbf{cns} = e_i.\mathbf{prd} = \mathbf{0}$. Such a path may be composed by a finite cyclic path which can be traversed an infinite number of times, or in the case of an infinite dynamic state transition diagram in fact by an infinite sequence of such edges. If such a path existed, the actor could indeed be activated an infinite number of times because of the required liveliness, but would possibly consume or produce no more tokens, again resulting in a deadlock of the entire model.*



(a) Path of edges in G_d.



(b) Equivalent composite edge.

Figure 7. Composite edge resulting from the merging of multiple consecutive edges.

Obviously, the diagram in Figure 6 meets both requirements because all states have at least one outgoing edge and all edges either consume one token from input port i_1 or produce one token on output port o_1.

Now, in order to be able to classify a larger set of actors into the *SDF* or *CSDF* models of computation, we need to *merge* the edges of a path $(e_1, e_2, \ldots, e_n)$ in the supplied state transition diagram into a so-called *composite edge* so that the vectors of tokens consumed and produced by this composite edge equals the vectors given by the functions $\mathbf{cns}(\rho)$ and $\mathbf{prd}(\rho)$ for a specific phase ρ, $1 \leq \rho \leq \tau(a)$:

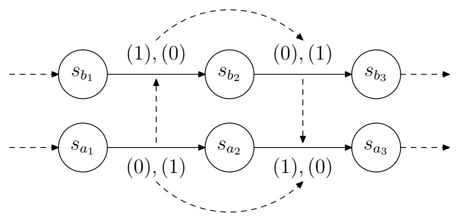
$$\left(\sum_{i=1}^{n} e_i.\mathbf{cns} = \mathbf{cns}(\rho) \right) \wedge \left(\sum_{i=1}^{n} e_i.\mathbf{prd} = \mathbf{prd}(\rho) \right)$$

In the following, an *activation* (or *firing*) of the actor a always means the atomic execution of all edge transitions belonging to such a composite edge.

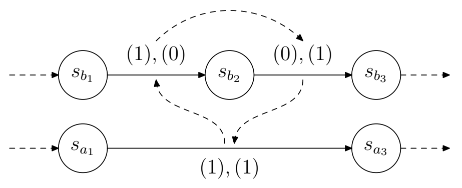
Figure 7 gives an example of an *SDF* actor with $\mathbf{cns}(1) = \mathbf{prd}(1) = (2)$ (i.e., each actor activation consumes two tokens from the input port i_1 and produces two tokens on the output port o_1): As described above, one such actor activation can consist of multiple consecutive transitions which are executed as a single transition that consumes and produces exactly the number of tokens required, also shown in Figure 7.

However, we require that all tokens produced by such a composite edge are dependent from the tokens consumed by the same composite edge, i.e.:

$$\exists k, 1 \leq k \leq n :$$
$$\left(\sum_{i=1}^{k} e_i.\mathbf{cns} = \mathbf{cns}(\rho) \right) \wedge \left(\sum_{i=k}^{n} e_i.\mathbf{prd} = \mathbf{prd}(\rho) \right)$$



(a) Feedback loop between two actors a and b.



(b) Cyclic dependency after the merging of edges.

Figure 8. The dashed arrows represent dependencies between the corresponding transitions. The merging of edges in a feedback loop (a) may lead to a cyclic dependcy (b), thus introducing deadlocks.

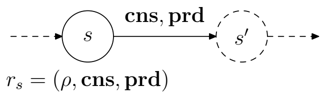
Otherwise, the merging of edges that are part of a feedback loop over other actors (i.e., tokens produced by an actor a trigger the production of tokens for a in other actors) may lead to a cyclic dependcy, thus introducing deadlocks into the model.

Figure 8 gives an example of such a feedback loop and the resulting cyclic dependency after merging both edges $(s_{a_1}, (0), (1), s_{a_2})$ and $(s_{a_2}, (1), (0), s_{a_3})$ of the dynamic state transition diagram into the composite edge $(s_{a_1}, (1), (1), s_{a_3})$. It should be noted that this restriction could be relaxed after the analysis of feedback loops. In the following, however, we assume always the worst case, i.e., all tokens produced by the activation of a in a specific phase depend on the tokens consumed by the activation of a in the same phase.

In the following, we will name the tuple $(\tau(a), \mathbf{cns}, \mathbf{prd})$ a *classification candidate* c, consisting of the number of phases $\tau(a)$ and the above defined functions $\mathbf{cns}$ and $\mathbf{prd}$. However, we will continue to refer directly to the members of c without using the '.'-operator.

Validation of classification candidates The following classification algorithm consists of two parts. First, we will show how a specific classification candidate c can either be *accepted* or *discarded*. Subsequently, we will discuss how to *construct* these classification candidates.

Comparison operations between two vectors of size k will be defined as follows: $\mathbf{a} \geq \mathbf{b} \Leftrightarrow a_i \geq b_i, 1 \leq i \leq k$, $\mathbf{a} > \mathbf{b} \Leftrightarrow \mathbf{a} \geq \mathbf{b} \wedge \mathbf{a} \neq \mathbf{b}$, $\mathbf{a} \leq \mathbf{b} \Leftrightarrow a_i \leq b_i, 1 \leq i \leq k$ and $\mathbf{a} < \mathbf{b} \Leftrightarrow \mathbf{a} \leq \mathbf{b} \wedge \mathbf{a} \neq \mathbf{b}$.



$\mathbf{cns}, \mathbf{prd}$
s
s'
$r_s = (\rho, \mathbf{cns}, \mathbf{prd})$

Figure 9. A tuple r_s is assigned to a visited state $s \in S$ containing the current phase ρ and the vectors $\mathbf{cns}$ and $\mathbf{prd}$ of tokens yet to be consumed and produced in that phase.

In order to validate c, a tuple $r_s = (\rho, \mathbf{cns}, \mathbf{prd})$ will be assigned to each visited state $s \in S$ which contains the vectors of tokens yet to be consumed and produced in phase $r_s.\rho$, i.e., $\mathbf{0} \leq r_s.\mathbf{cns} \leq \mathbf{cns}(r_s.\rho)$ and $\mathbf{0} \leq r_s.\mathbf{prd} \leq \mathbf{prd}(r_s.\rho)$, respectively. This concept is schematically depicted in Figure 9.

In the following, we will refer to the set of outgoing edges of an actor state $s \in S$ in the supplied state transition diagram as $E_s \subseteq E$, i.e., $E_s = \{ e \in E \mid e.s = s \}$.

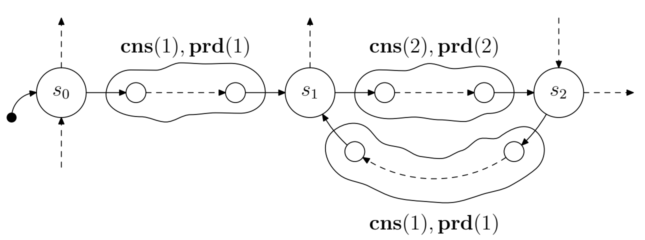
Not all annotations of actor states with tuples are valid ones. Several restrictions can be derived from the number of tokens consumed and produced by the outgoing edges E_s of an actor state $s \in S$:

Definition 8 (Edge criteria I) *Each outgoing edge $e \in E_s$ of a visited actor state $s \in S$ must consume and produce less or equal tokens than specified by the annotated tuple r_s, i.e., $\forall e \in E_s : r_s.\mathbf{cns} \geq e.\mathbf{cns} \wedge r_s.\mathbf{prd} \geq e.\mathbf{prd}$. Otherwise, the annotated tuple r_s is invalid.*

Definition 9 (Edge criteria II) *Each outgoing edge $e \in E_s$ of a visited actor state $s \in S$ must not produce tokens if there are still tokens to be consumed in that phase after the execution of the corresponding transition of e, i.e., $\forall e \in E : r_s.\mathbf{cns} = e.\mathbf{cns} \vee e.\mathbf{prd} = \mathbf{0}$. Otherwise, the annotated tuple r_s is invalid.*

Assuming the visited actor state $s \in S$ is annotated with a valid tuple r_s, we can calculate for each edge $e \in E_s$ the *expected* tuple $r'_{e.s'}$ to be assigned to the next actor state $e.s'$ as follows: If $r_s.\mathbf{cns} = e.\mathbf{cns}$ and $r_s.\mathbf{prd} = e.\mathbf{prd}$, all remaining tokens of phase $r_s.\rho$ are consumed and produced after the execution of the corresponding transition of e, thus $r'_{e.s'} = (r_s.\rho + 1, \mathbf{cns}(r_s.\rho + 1), \mathbf{prd}(r_s.\rho + 1))$ (if $r_s.\rho < \tau(a)$) or $r'_{e.s'} = (1, \mathbf{cns}(1), \mathbf{prd}(1))$ (if $r_s.\rho = \tau(a)$). Otherwise, some tokens remain to be consumed or produced in phase $r_s.\rho$ after the execution of the corresponding transition of e, thus $r'_{e.s'} = (r_s.\rho, r_s.\mathbf{cns} - e.\mathbf{cns}, r_s.\mathbf{prd} - e.\mathbf{prd})$.

Let $s \in S$ be an actor state for which an expected tuple r'_s has been calculated. The following cases may occur: If s has not been visited yet, it will be marked as visited and annotated with r'_s, i.e., $r_s = r'_s$. If s has been previously visited and the annotated tuple r_s is not equal to the expected tuple r'_s, some of the already annotated tuples must be inconsistent, and no further states and edges will



$\mathbf{cns}(1), \mathbf{prd}(1)$
$\mathbf{cns}(2), \mathbf{prd}(2)$
s_0
s_1
s_2
$\mathbf{cns}(1), \mathbf{prd}(1)$

Figure 10. Assuming an actor a exhibits *CSDF* behavior with $\tau(a) = 2$, any path from the initial state s_0 must comply with the functions $\mathbf{cns}$ and $\mathbf{prd}$.

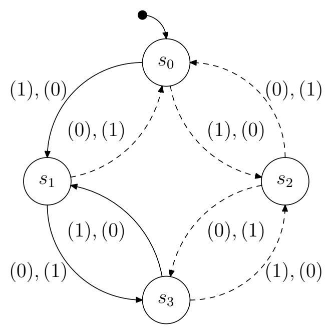
be processed. Otherwise, the already annotated tuples are consistent, and further states and edges will be processed.

The first actor state which can be annotated in this way is the initial actor state $s_0 \in S$. All *SDF* and *CSDF* actors are initially in phase 1, so that $r_{s_0}.\rho = 1$. Also, no tokens could have been consumed or produced for that phase in this initial state yet. Therefore, $r_{s_0}.\mathbf{cns} = \mathbf{cns}(1)$ and $r_{s_0}.\mathbf{prd} = \mathbf{prd}(1)$. The initial state is then marked as visited and the outgoing edges can be checked against Definition 8 and Definition 9, etc. In this way, a breadth-first search is established which either accepts or discards the classification candidate c. Therefore, the time complexity is $O(|S| + |E|)$.

Construction of classification candidates This section describes the systematic construction of the classification candidates c required for the algorithm presented previously. First, it can be observed that all paths from s_0 must comply with the assumed parameters $\tau(a)$, $\mathbf{cns}$ and $\mathbf{prd}$ if the actor has *SDF* or *CSDF* semantics. Otherwise, the chosen classification candidate c would be discarded. Therefore, one can construct the classification candidates c by analyzing a *single arbitrary path* $p = (e_1, e_2, \ldots, e_n)$ from the initial state $s_0 \in S$ (i.e., $e_1.s = s_0$). Figure 10 shows an example for such a path.

In principle, the number of edges in p can be infinite due to Definition 6. However, one can stop adding edges to p if an actor state is reached which is already traversed by p. In this case, a cyclic subpath of p exists, which could be traversed an infinite number of times and therefore must also comply with the assumed parameters $\tau(a)$, $\mathbf{cns}$ and $\mathbf{prd}$. Such a cyclic subpath can also be seen in Figure 10. In the following, a path p with these properties is called a *candidate path*.

Definition 10 (Candidate path) *A* candidate path $p = (e_1, e_2, \ldots, e_n)$ *is a path in the supplied state transition diagram with* $e_1.s = s_0$ *which can be decomposed into an acyclic prefix path* p_a *and a cyclic path* p_c *such that* $p = p_a{}^\frown p_c$, *i.e.,* $\exists i, 1 \leq i \leq n : e_n.s' = e_i.s$ *and* $\forall j, 1 \leq j < n : \nexists i, 1 \leq i \leq j : e_j.s' = e_i.s$.



$s \in S$	$(s_{\text{func}}, s_{\text{fsm}})$
s_0	$((0,0), s)$
s_1	$((1,0), s)$
s_2	$((0,1), s)$
s_3	$((1,1), s)$

(a) G_d of actor a from Figure 5. (b) The actor state space.

Figure 11. A possible candidate path p is represented by the non-dashed arrows in (a).

It may be noted that some edges contained in such a candidate path p must consume and produce tokens due to Definition 7. In particular, this holds true for the cyclic subpath of p.

The length of p is bounded by the number of states $|S|$: Let $p' = (e_1, e_2, \ldots, e_n)$ be a path of length $n = |S| - 1$ which connects all states of the supplied state transition diagram but contains no cyclic subpath, i.e., $\forall j, 1 \leq j \leq n : \nexists i, 1 \leq i \leq j : e_j.s' = e_i.s$. Due to the required liveliness there must exist at least one outgoing edge from state $e_n.s'$, i.e., $E_{e_n.s'} \neq \emptyset$. Let $e \in E_{e_n.s'}$ be an arbitrary edge of these outgoing edges and $p = p'{}^\frown(e)$ the concatenation of p' with edge e. Due to p''s traversing of all states, $e.s' \in S$ must also have been visited by p'. Therefore p is a candidate path with the length $|S| - 1 + 1 = |S|$ satisfying Definition 10.

With the help of a candidate path p, the classification candidates c for a state transition diagram can be constructed by successively merging consecutive edges of p into composite edges as follow: Let $e_i, 1 \leq i \leq n$ be the first edge of p that consumes or produces some tokens, i.e., $\forall k, 1 \leq k < i : e_k.\mathbf{cns} = e_k.\mathbf{prd} = 0$ and $e_i.\mathbf{cns} \neq 0 \vee e_i.\mathbf{prd} \neq 0$. This edge must exist because of Definition 7. The first classification candidate therefore evaluates to $c_1 = (1, (e_i.\mathbf{cns}), (e_i.\mathbf{prd}))$.

If c_1 is accepted, a is an *SDF* actor with the corresponding parameters. Otherwise, one needs to calculate the next classification candidate c_2. This is accomplished by finding the next edge $e_j, i < j \leq n$ of p that consumes or produces some tokens, i.e., $\forall k, i < k < j : e_k.\mathbf{cns} = e_k.\mathbf{prd} = 0$ and $e_j.\mathbf{cns} \neq 0 \vee e_j.\mathbf{prd} \neq 0$. This edge does not exist if $i = n$ or $\forall k, i < k \leq n : e_k.\mathbf{cns} = e_k.\mathbf{prd} = 0$. In this case, a is neither an *SDF* nor an *CSDF* actor (because c_1 was discarded). If a valid edge e_j was found, two cases must be distinguished: If tokens are produced by e_i and consumed by e_j, a new phase has to be appended to the first phase, i.e., $c_2 = (2, (e_i.\mathbf{cns}, e_j.\mathbf{cns}), (e_i.\mathbf{prd}, e_j.\mathbf{prd}))$. Otherwise, the vectors of consumed and produced tokens will be added to those of the first phase, i.e., $c_2 = (1, (e_i.\mathbf{cns} + e_j.\mathbf{cns}), (e_i.\mathbf{prd} + e_j.\mathbf{prd}))$.

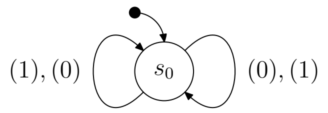


Figure 12. Reduced state transition diagram G_r of the actor shown in Figure 5.

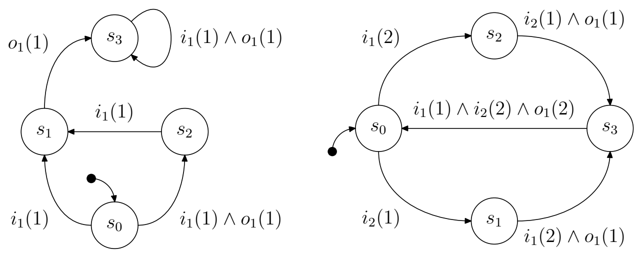
Now, c_2 is validated, etc. The complexity of this algorithm depends on the length of the candidate path p: In the worst case, n different classification candidates may be constructed. As has already been shown, $n \leq |S|$. Therefore, the overall worst case complexity evaluates to $O(|S|(|S|+|E|)) = O(|S|^2 + |S||E|)$.

Figure 11 shows again the dynamic state transition diagram from Figure 6. A possible candidate path is, e.g., $p = ((s_0,(1),(0),s_1),(s_1,(0),(1),s_3),(s_3,(1),(0),s_1))$, resulting in the first classification candidate $c_1 = (1,((1)),((0)))$. The annotated tuple for s_0 is therefore $r_{s_0} = (1,(1),(0))$. Checking the outgoing edges of s_0 against Definition 8 and Definition 9 reveals no problems. Thus, the expected tuples for s_1 and s_2 can be calculated and assigned to these (unvisited) states, i.e., $r_{s_1} = r_{s_2} = (1,(1),(0))$. However, checking the outgoing edges of s_1 and s_2 discards c_1 because of Definition 8: One token is produced by each edge, but the annotated tuples suggest that there are no tokens remaining to be produced in this phase. The next classification candidate is $c_2 = (1,((1)+(0)),((0)+(1)) = (1,((1)),((1)))$ because no tokens were produced in this phase yet. In contrast to c_1, c_2 is accepted by the above algorithm. Therefore, the actor from Figure 5 is an *SDF* actor which consumes one token from i_1 and produces one token on o_1 in each activation.

Reduced state transition diagram As has been already discussed, the dynamic state transition diagram may be infinite, hence being intractable by the algorithm. Furthermore, the functionality state of an actor may not be known in advance, resulting in further restrictions to be introduced for a working implementation of the classification algorithm.

To abstract from the functionality state of the actor, we consider a state transition diagram which is basically equivalent to the *FSM* of the actor, with the exception that transitions from the same *FSM* state which consume and produce the same number of tokens result in only one edge being added to the set of edges:

Definition 11 (Reduced state transition diagram) *The* reduced state transition diagram *of an actor a is the tuple $G_r = (S, E, s_0)$ containing the set of states $S = \mathcal{R}.S_{fsm}$ and the initial state $s_0 = \mathcal{R}.s_{0,fsm} \in S$. The set of edges E is defined and constructed as in Definition 5, with the exception that $\delta(s)$ is used as the set of outgoing transitions instead of $\epsilon(s)$.*



(a) *SDF* actor (b) *CSDF* actor

Figure 13. *FSM*s recognized as (a) *SDF* with $\mathbf{cns}(1) = \mathbf{prd}(1) = (1)$, and (b) *CSDF* with $\mathbf{cns}(1) = (2,1)$, $\mathbf{cns}(2) = (1,2)$, $\mathbf{prd}(1) = (1)$ and $\mathbf{prd}(2) = (2)$.

The classification algorithm can always be applied to such a reduced state transition diagram because the set of *FSM* states must be finite in contrast to the actor state space which may be infinite. However, some actors will not be classified into the *SDF* or *CSDF* models of computation any more in spite of the dynamic state transition diagram showing a static communication behavior.

Figure 12 shows the reduced state transition diagram G_r of the actor from Figure 5. The classification algorithm classifies this diagram neither into the *SDF* nor *CSDF* model of computation, although it was previously shown that indeed an equivalent *SDF* actor does exist.

Finally, Figure 13 shows two actor *FSM*s whose reduced state transition diagrams were automatically constructed and subsequently analyzed by the classification algorithm.

4. Limitations

Due to the general undecidable nature of the problem the presented classification algorithm provides only a sufficient but not necessary condition for a static data flow actor. A program to be analyzed must meet certain restrictions, e.g. loops must have a bounded number of iterations, no recursive or dynamic function calls etc. In particular, the classification algorithm lacks propagation of guard condition invariants across firing steps, e.g., even if we know that a certain guard has evaluated to true to take a code path and this path does not change the guard value, the next evaluation of this guard in the code path will again be considered uncertain. However, assuming that each code path which does not contain communication operations is finite all presented limitations only apply to code paths containing communication operations. This stems from the irrelevance of code paths implementing the actor functionality to the communication behavior of the actor and therefore to the classification of the actor into a static data flow model of computation.

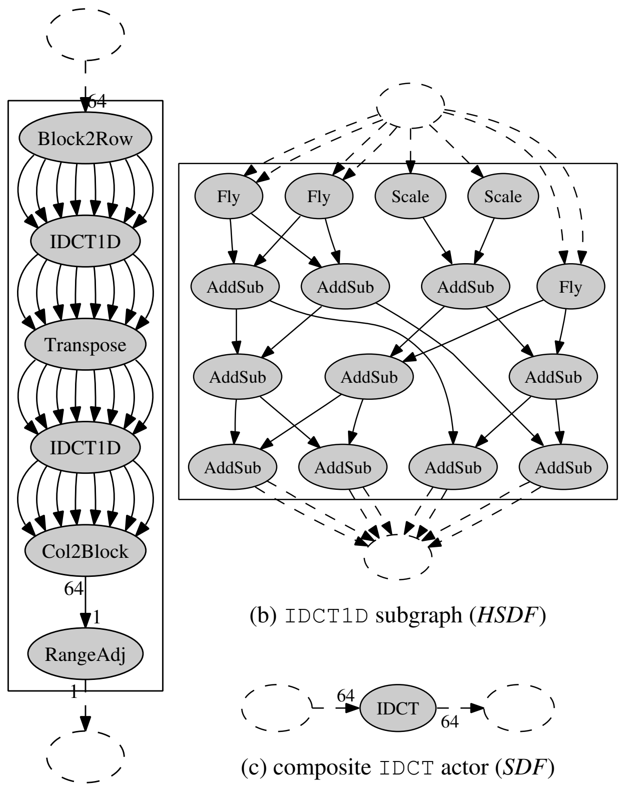
5. Related Work

Models of Computation (*MoC*s) are an important concept in the design of embedded systems [14]. *MoC*s permit the use of efficient domain-specific optimization methods [19]. The advantages have been shown by many examples, e.g., for real time reactive systems [2] and in the signal processing domain [4, 3]. An environment for modeling and simulating different and heterogeneous *MoC*s is provided by Ptolemy II [12]. Actors are classified by the domain they are assigned to and the domain is explicitly stated by a so called director that is responsible for proper actor invocation. In Ptolemy II, heterogeneous *MoC*s can be composed hierarchically, i.e. an actor can be refined by a network of actors again controlled by a domain-specific director.

Other heterogeneous *MoC*s have been proposed in literature. FunState (Functions driven by state machines) [18] is of particular interest as it is similar to our proposed model. However, the authors did only provide some kind of modeling guidelines to translate static and dynamic data flow models as well as finite state machines into FunState. The reverse, i.e., the actor classification, was neglected.

On the other hand, SystemC [8, 1] is becoming the de facto standard for the design of digital hardware/software systems. SystemC permits the modeling of many different *MoC*s, but there is no unique representation of a particular *MoC* in SystemC. In order to identify different *MoC*s in SystemC, Patel et al. [16] have extended SystemC itself with different simulation kernels for *Communicating Sequential Processes* and *Finite State Machine MoCs* which also improves the simulation efficiency. Thus, the classification is again based on explicit statements. Other approaches supporting to model well-defined *MoC*s are also library-based and do not support actor classification, e.g., *YAPI* [6] and *SHIM* [7]. Thus, to the best of our knowledge, no publications on *MoC* classification for SystemC designs do exist today. This is not surprisingly as this problem in its general form is not decidable.

One major problem, also when restricting to a subset of SystemC, is the unstructured use of communication primitives which makes the automatic classification of a SystemC design a hard problem. The SystemC Transaction Level Modeling (*TLM*) standard [17] does not alleviate these problems because it is not concerned with defining representations of *MoC*s in SystemC. Instead, the *TLM* standard defines transaction level interfaces via method calls, therefore improving simulation efficiency and providing the foundation for platform-based design in SystemC. The work of Habibi et al. [10, 9] and Niemann and Haubelt [15] goes one step further. They use the *TLM* standard in combination with (Abstract) Finite State Machines which specify the behavior of single SystemC modules. In both cases, the focus is on the formalization of the entire SystemC transaction level model, and not on the classification of a single actor.



(a) hierarchical `IDCT` actor

(b) `IDCT1D` subgraph (*HSDF*)

(c) composite `IDCT` actor (*SDF*)

Figure 14. Composite `IDCT` actor (c) resulting from clustering all static data flow actors of (a) and (b).

6. Results

In order to evaluate the optimization potential for real-world examples, we have applied the classification algorithm to our Motion-*JPEG* decoder from Figure 1. As a result, the `JPEGSource` actor, the `InvQuant` actor, the `InvZigZag` actor, and additionally all actors of the hierarchical `IDCT` actor were classified either into the *SDF* or *CSDF* model of computation. The hierarchical `IDCT` actor transforms blocks of 8×8 frequency coefficients into equally sized image blocks, and is depicted in detail in Figure 14(a) and (b).

This allows us to perform optimization steps including the clustering of the static data flow actors into a single so-called *composite actor* shown in Figure 14(c) by statically scheduling all actors from Figure 14(a) and (b). This composite `IDCT` actor must, however, be dynamically scheduled due to the heterogeneous environment in which it is embedded (cf. Figure 1): To determine if the composite

IDCT actor could be activated, both the incoming and outgoing *FIFO* channel must be checked for the required number of tokens and free space.

Without knowing the underlying model of computation, this optimization step would not be possible, i.e., no composite actor could be generated and the IDCT actor must be synthesized as seen in Figure 14(a) and (b). Therefore, instead of having to check the fill level of only two *FIFO* channels, the dynamic scheduler must check all 55 *FIFO* channels in turn, wasting precious computational power. Moreover, besides checking the fill level of the *FIFO* channels, the dynamic round-robin scheduler must also poll each actor individually to see if it can fire or not, due to the guards defined by the user.

Comparing the latencies of the Motion-*JPEG* decoder, the version containing the composite IDCT outperforms the version implementing the hierarchical IDCT actor approximately by a factor of two, as can be seen in the following table:

hier. IDCT	comp. IDCT	Reduction
0.305s	0.132s	57%

Due to the single-processor scheduling, the throughput could only be improved by the same value of 57%.

7. Conclusions

Applications in the signal processing domain are often modeled by data flow graphs. Due to the complex nature of today's systems these applications contain both dynamic and static parts. In this paper, we have proposed a formal notation encompassing both of these extremes, while still being able to classify an actor from its given formal notation into the synchronous or cyclo-static data flow domain. This enables us to use a unified descriptive language to express the behavior of actors while still retaining the advantage to apply domain-specific optimization methods to parts of the system. Our classification approach allows us to identify the static parts of a given heterogeneous data flow graph derived from a SystemC model. The information about static subgraphs of the data flow graph may be exploited by applying static scheduling to these parts, therefore improving latency and throughput of the modeled Motion-*JPEG* application by 57%. In future work we will extend the set of recognized static data flow actors by also considering invariants of guard conditions across firing steps.

References

[1] M. Baird, editor. *IEEE Standard 1666-2005 SystemC Language Reference Manual*. IEEE Standards Association, New Jersey, USA, 2005.

[2] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, U. Freund, E. Schlenker, and H.-J. Wolff. Correct-by-Construction Transformations across Design Environments for Model-Based Embedded Software Development. In *Proceedings of DATE*. IEEE Computer Society, Mar. 2005.

[3] B. Bhattacharya and S. Bhattacharyya. Parameterized Dataflow Modeling of DSP Systems. In *Proceedings of ICASSP*, pages 1948–1951, June 2000.

[4] S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for signal processing systems. *IEEE Transactions on Circuits and Systems*, 47(9), Sept. 2000.

[5] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *IEEE Transaction on Signal Processing*, 44(2):397–408, Feb. 1996.

[6] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI: Application Modeling for Signal Processing Systems. In *Proceedings of DAC*, pages 402–405, June 2000.

[7] S. A. Edwards and O. Tardieu. SHIM: A Deterministic Model for Heterogeneous Embedded Systems. In *Proceedings of EMSOFT*, pages 264–272, 2005.

[8] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[9] A. Habibi, H. Moinudeen, and S. Tahar. Generating Finite State Machines from SystemC. In *Proceedings of DATE*, pages 76–81. IEEE Computer Society, Mar. 2006.

[10] A. Habibi, S. Tahar, A. Samarah, D. Li, and O. A. Mohamed. Efficient Assertion Based Verification using TLM. In *Proceedings of DATE*, pages 106–111. IEEE Computer Society, Mar. 2006.

[11] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[12] E. A. Lee. Overview of the ptolemy project, technical memorandum no. ucb/erl m03/25. Technical report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, 94720, USA, July 2004.

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

[14] E. A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, Dec. 1998.

[15] B. Niemann, C. Haubelt, M. Uribe, and J. Teich. Formalizing TLM with Communicating State Machines. In S. A. Huss, editor, *Advances in Design and Specification Languages for Embedded Systems*, pages 225–242. Springer, 2007.

[16] H. D. Patel and S. K. Shukla. Towards a heterogeneous simulation kernel for system-level models: a SystemC kernel for synchronous data flow models. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 24, pages 1261–1271, Washington, D.C., Aug. 2005.

[17] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez. Transaction level modeling in SystemC. 2004.

[18] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. FunState - An Internal Design Representation for Codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):524–544, Aug. 2001.

[19] D. Ziegenbein, K. Richter, R. Ernst, L. Thiele, and J. Teich. SPI- a system model for heterogeneously specified embedded systems. *IEEE Trans. on VLSI Systems*, 2002.