

# Analysis of SystemC Actor Networks for Efficient Synthesis

## Submitted to the Special issue on Model-driven Embedded System Design

JOACHIM FALK, CHRISTIAN ZEBELEIN, JOACHIM KEINERT,  
CHRISTIAN HAUBELT and JUERGEN TEICH

University of Erlangen-Nuremberg

and

SHUVRA S. BHATTACHARYYA

University of Maryland

---

Applications in the signal processing domain are often modeled by data flow graphs. Due to heterogeneous complexity requirements, these graphs contain both dynamic and static data flow actors. In previous work, we presented a generalized clustering approach for these heterogeneous data flow graphs in the presence of unbounded buffers. This clustering approach allows the application of static scheduling methodologies for static parts of an application during embedded software generation for multi-processor systems. It systematically exploits the predictability and efficiency of the static data flow model to obtain latency and throughput improvements. In this paper, we present a generalization of this clustering technique to data flow graphs with bounded buffers, therefore enabling synthesis for embedded systems without dynamic memory allocation. Furthermore, a case study is given to demonstrate the performance benefits of the approach.

Categories and Subject Descriptors: D.1.3 [Software]: Programming Techniques—*Concurrent Programming - Parallel Programming*

General Terms: Clustering algorithm

Additional Key Words and Phrases: Data flow analysis, actor-oriented design, clustering, scheduling

---

### 1. INTRODUCTION

At Electronic System Level (ESL), a trend towards *actor-oriented programming models* [Jerraya et al. 2006; Edwards and Tardieu 2005] can be identified. For instance, data flow models are well suited to model multimedia applications. As an example, consider the top level data flow model of a Motion-*JPEG* decoder (depicted in Figure 1) which has been developed in SystemC [Grötke et al. 2002; Baird 2005]. It contains both static data flow actors (shaded vertices), i.e., actors with constant consumption and production rates known from *synchronous data flow (SDF)* models [Lee and Messerschmitt 1987] and *cyclo-static data flow (CSDF)* models [Bilsen et al. 1996], as well as *dynamic data flow (DDF)* actors like the *Parser* which is modeled by a *Kahn process* [Kahn 1974].

When synthesizing such a data flow graph, a scheduling strategy has to be selected. While many research groups have conducted research on optimally mapping data flow graphs onto a given MPSoC platform, e.g., Daedalus [Thompson

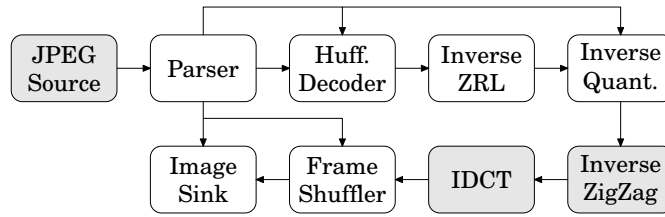


Fig. 1. Example of a Motion-*JPEG* decoder including dynamic as well as static data flow actors (shaded). Communication between actors (vertices) is realized via *FIFOs* (directed edges).

et al. 2007], SystemCoDesigner [Keinert et al. 2009], Koski [Kangas et al. 2006], System-on-Chip Environment (SCE) [Abdi et al. 2003], etc., there is still relatively little work on scheduling of heterogeneous data flow graphs. On the other hand, *static scheduling* for multi-processor systems is solved for models with limited expressiveness, e.g., static data flow graphs. For instance, efficient single processor scheduling algorithms exist for *SDF* models [Bhattacharyya et al. 1995; Hsu and Bhattacharyya 2007]. Unfortunately, these algorithms are constrained to pure *SDF* graphs while real world multimedia applications not only require to be modeled via *heterogeneous* data flow graphs which contain actors with higher levels of complexity, but are also often embedded in a control oriented and event driven environment which interacts with or (re)configures the actors [Geilen and Basten 2004]. The most basic strategy is to postpone all scheduling decisions to runtime (*dynamic scheduling*) with the resulting significant scheduling overhead and, hence, a reduced system performance. However, this strategy is suboptimal if we know that some of the actors exhibit regular communication behavior like *SDF* and *CSDF* actors. The scheduling overhead problem could be mended by coding the actors of the application at an appropriate level of granularity, i.e., combining so much functionality into one actor that the computation costs dominate the scheduling overhead. This can be thought of as the designer manually making clustering decisions. However, the appropriate level of granularity is chosen by the designer by trading schedule efficiency improving bandwidth against resource consumption (e.g., buffer capacities), and scheduling flexibility improving latency. Combining functionality into one actor can also mean that more data has to be produced and consumed atomically which requires larger buffer capacities and may delay data unnecessarily, therefore degrading the latency if multiple computation resources are available for execution of the application. Furthermore, if the mapping of actors to resources itself is part of the synthesis step, e.g., design space exploration [Keinert et al. 2009], an appropriate level of granularity can no longer be chosen in advance by the designer as it depends on the mapping itself.

In real world designs actors are modeled using a specification language that is expressive enough to specify the most general data flow actors, e.g., SystemC. Hence, information about actors belonging to a restricted class of models of computation is not available and a preprocessing step is required to perform the classification of actors to derive the missing information. This information is the necessary basis for all other algorithms presented in this paper. In previous work [Zebelein et al. 2008]

we presented such a preprocessing step to classify a restricted subset of SystemC actors into different kinds of data flow actors. The two steps necessary for this classification are (i) extraction of the *CFG* for each SystemC actor, presented in Section 2, which is transformed into the formal model, given in Section 3, and (ii) a classification step, reviewed in Section 4, which decides for each actor based on the derived model its corresponding actor category from the three categories of interest in this paper, i.e., *SDF*, *CSDF*, or *DDF*. As the problem of actor classification in its general form is undecidable the methodology presented in [Zebelein et al. 2008] is only a sufficient criterion. Actors which can neither be classified as *SDF* nor *CSDF* will be treated as *DDF* actors.

In order to obtain efficient schedules, one solution could be the replacement of the static data flow subgraph by a single actor, i.e, clustering all static data flow actors into a new actor. Unfortunately, existing algorithms might result in infeasible schedules or greatly restrict the clustering design space by considering only *SDF* subgraphs that can be clustered into monolithic *SDF* actors without introducing deadlock. The problem stems from the fact that static scheduling adds constraints to the possible schedules of a system. Imposing too strict constraints on the possible schedules excludes all feasible schedules deadlocking the system.

In previous work [Falk et al. 2008], we generalized clustering to produce a dynamic *composite actor* from a static data flow subgraph in the presence of infinite buffers. This composite actor implements a *quasi-static schedule* by means of a so-called *cluster FSM*. The composite actor is constructed in such a way that it always produces a maximum number of output tokens from a minimum number of input tokens, i.e., the composite actor will never require more input tokens to produce an output token than are consumed by the replaced subgraph on the channels crossing the border between the environment and itself to produce the equivalent output token. The basic construction of the composite actor will be reviewed in more detail in Section 5.

The clustering methodology presented in [Falk et al. 2008] is limited in a way that it does not handle *FIFOs* with bounded capacities and is therefore unsuitable for code generation for embedded systems supporting only static memory allocation. In this paper, we present a generalization of this clustering technique to data flow graphs with bounded *FIFO* capacities, therefore enabling synthesis for embedded systems without dynamic memory allocation. The generalization is based on the observation that most data flow graphs do not change their behavior if buffer sizes are increased. The presented algorithms assumes this property as well as a deadlock free data flow graph annotated with buffer bounds as input. These annotations are required as it is in general undecidable whether a Kahn process network can be scheduled within bounded memory [Buck 1993]. This enables us to calculate an upper bound adjustment of the buffer capacities to retain the deadlock free property of the data flow graph to handle the constraints induced by buffer limitations. Furthermore, modeling a bounded *FIFO* with two unbounded *FIFOs* is suboptimal in the sense that it would give data dependencies (handled in [Falk et al. 2008]) the same priority as dependencies due to buffer limitations. In general, the more complex the dependencies between consumed input tokens and produced output tokens become, the more complex the schedule becomes and therefore the more

performance is lost due to scheduling overhead. However, dependencies imposed by buffer limitations can be circumvented by enlarging the buffers while data dependencies have no analogous solution. The buffer adjustment is presented in Section 6. Subsequently we discuss results in Section 7. Finally, related work and conclusions will be presented in Section 8 and Section 9.

## 2. COMMUNICATION EXTRACTION

SystemC [Grötter et al. 2002; Baird 2005] has been chosen as entry for our design flow due to its usage as real world *system description language* for industrial projects. However, due to its usage of C++ it inherits all the problems of C++, e.g., the possibility of unstructured communication over shared memory and Turing completeness. Fortunately, SystemC mandates a certain way of design representation, i.e., SystemC strongly suggests to represent a design in an *actor-oriented* way [Jerraya et al. 2006; Edwards and Tardieu 2005]. Designs represented in an actor-oriented way consist of *actors* which can only communicate via *channels* connecting the actors via *ports*, hence alleviating the unstructured communication problem inherited from C++. Furthermore, we constrain the actors to only communicate over channels exhibiting *FIFO* semantics. In practice, we use `sc_fifo` SystemC primitives to implement channels of the desired *FIFO* semantics. An example of a SystemC module satisfying our constraints is depicted in Figure 2.

```
class FrameShuffler: public sc_module {
public:
    sc_fifo_in<uint8_t>    in;
    sc_fifo_in<CtrlToken> c;
    sc_fifo_out<Pixel>    out;
protected:
    static const size_t SHUFFLE_LINES = 8;

    void shuffle();
public:
    SC_HAS_PROCESS(FrameShuffler);

    FrameShuffler(sc_module_name name): sc_module(name)
        { SC_THREAD(shuffle); }
};
```

Fig. 2. Example of a SystemC module implementing the `FrameShuffler` vertex from Figure 1.

In order to simplify our clustering algorithm description we need a model of an actor which abstracts away all the unnecessary details. Briefly, we discard the data path of an actor and only work with the *actor FSM* which describes the *communication behavior* of the actor. This abstraction can be performed due to the data agnostic approach of clustering, i.e., clustering only cares about how much data is consumed and produced but does not care what data values are consumed or produced. More formally, an *actor FSM* is defined as follows:

**Definition 2.1 (Actor FSM)** The *actor FSM* is a tuple  $\mathcal{R} = (S_{\text{fsm}}, s_0, T, \text{cons}, \text{prod}, G_{\text{guard}}, F_{\text{action}})$  containing: A set of *states*  $S_{\text{fsm}}$ . An *initial state*  $s_0 \in S_{\text{fsm}}$ . A set of *transitions*  $(s, s') \in T \subseteq S_{\text{fsm}} \times S_{\text{fsm}}$  containing the current state  $s$  and the next state  $s'$ . A function  $\text{cons} : T \times I \rightarrow \mathbb{N}_0$  denoting the number of tokens  $\text{cons}(t, i)$  which must be available on input port  $i$  for the execution of transition  $t$ . A function  $\text{prod} : T \times O \rightarrow \mathbb{N}_0$  denoting the amount of free token places  $\text{prod}(t, o)$  which must be available on output port  $o$  for the execution of transition  $t$ . A guard mapping function  $G_{\text{guard}} : T \rightarrow \mathcal{F}$  mapping a transition  $t$  to a guard function  $g_{\text{guard}} = G_{\text{guard}}(t)$  evaluating to **true** or **false** which represents more general predicates depending on the state of the data path of the actor or the token values on the input ports. And an action mapping function  $F_{\text{action}} : T \rightarrow \mathcal{F}$  mapping a transition  $t$  to an action  $f_{\text{action}} = F_{\text{action}}(t)$  in the data path.

The actions  $f_{\text{action}}$  implement the data path of the actor while the predicates  $\text{cons}/\text{prod}$ , in the following called *input/output pattern*, encode how much data is consumed/produced on actor input/output ports. The conjunction of all predicates  $\text{cons}(t, i)$ ,  $\text{prod}(t, o)$ , and  $G_{\text{guard}}(t)$  is called the *activation pattern* of a transition  $t$ . The execution of an actor is then divided into atomic *firing steps*, each of which consists of three phases: (i) checking for enabled transitions  $t$  of each actor satisfying their activation patterns, (ii) selecting one enabled transition  $t$  per actor and executing the corresponding action  $F_{\text{action}}(t)$ , and (iii) consuming and producing tokens as encoded by  $\text{cons}(t, i)$  and  $\text{prod}(t, o)$ .

In the following, we describe the algorithm to derive an actor *FSM* from a SystemC module. The extraction uses the *CFG* of a SystemC module which is extracted via the *AST* generated by the *KaSCPar* [FZI Research Center for Information Technology 2007] parser. This analysis requires that the SystemC module contains exactly one SystemC thread, i.e., exactly one `SC_THREAD(method)` is defined in the module's constructor and assumes that only functions in the SystemC module itself issue read or write operations on the ports of the module. All other code, e.g., standard library functions, global helper code, and code reachable via function pointers or virtual methods are assumed to have no access to the ports of the SystemC module which calls these functions. The *C++ RTTI* at runtime after the *elaboration phase* of SystemC.

The SystemC module depicted in Figure 2 and 3 implementing the `FrameShuffler` actor from Figure 1 satisfies these constraints. Furthermore, we assume that as long as the method specified by `SC_THREAD` does not return, the actor will eventually consume another input token or produce another output token. To exemplify the proposed algorithm we consider the SystemC code for the `shuffle` method as depicted in Figure 3.

The analysis starts by deriving the *control flow graph (CFG)* of this method, e.g., depicted in Figure 4 for the `shuffle` method. The *CFG* deviates from a conventional control flow graph as known from compiler design, since *basic blocks* are split such that all `read` statements in a basic block precede all `write` statements in the basic block. This requirement stems from the fact that the actions  $f_{\text{action}}$  will be assembled from the basic blocks and token consumption and production requirements of

```

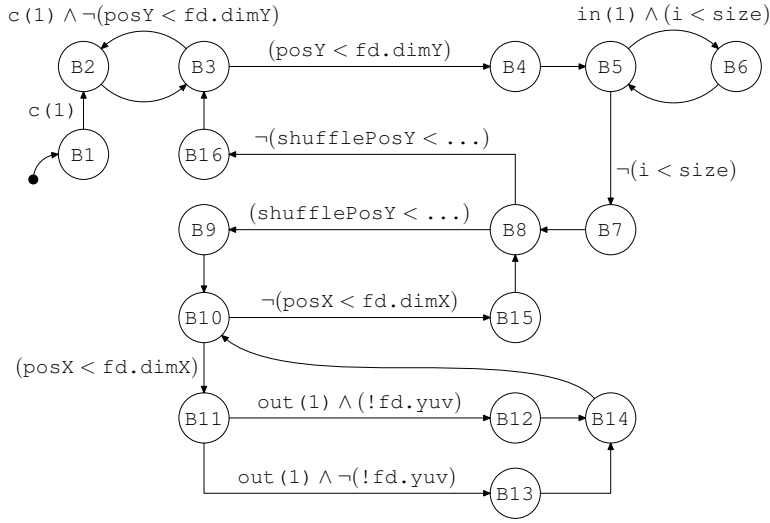
void FrameShuffler::shuffle() {
B1:  CtrlToken fd;
      while (true) {
B2:    c.read(fd);
        // Only support JPEG YCbCr 4:4:4 or Grayscale
        unsigned int size =
          (fd.yuv ? 3 : 1) * fd.dimX * SHUFFLE_LINES;
        uint8_t data[size];
        for (int posY = 0;
B3:          posY < fd.dimY;
B16:          posY += SHUFFLE_LINES) {
B4:          for (size_t i = 0;
B5:                i < size;
                    ++i)
B6:            in.read(data[i]);
B7:          for (int shufflePosY = 0;
B8:                shufflePosY < SHUFFLE_LINES;
B15:                ++shufflePosY) {
B9:            for (int posX = 0;
B10:                  posX < fd.dimX;
B14:                  ++posX) {
B11:              unsigned int posInBlock =
                (posX & 7U) + ((posY & 7U) << 3);
                unsigned int posOfBlock = ((posX & ~7U) << 3);
                if (!fd.yuv) { // Grayscale
B12:                  out.write(Pixel(
                    data[posInBlock | posOfBlock], 128, 128));
                } else { // YCbCr CbCr have a bias of 128
B13:                  out.write(Pixel(
                    data[posInBlock | (posOfBlock*3 + 0)],
                    data[posInBlock | (posOfBlock*3 + 64)],
                    data[posInBlock | (posOfBlock*3 + 128)]));
                }
            }
          }
        }
}

```

Fig. 3. Source code of the `shuffle` method from the `FrameShuffler` class as shown in Figure 2. Additionally the labels B1–B16 depict the *basic blocks* of the corresponding *control flow graph* (CFG).

an action are checked by the input/output pattern at the start of the action. If a basic block would contain a `read` statement after a `write` statement, an erroneous dependency from the token consumed by the `read` statement to the token produced by the `write` statement would be generated by the input/output pattern at the start of the action containing this basic block.

It should be noted that the CFG depicted in Figure 4 is already annotated with input/output patterns for each basic block, e.g., the input pattern `c(1)` on the edge between B1 and B2 denotes that one token must be available on the *FIFO* channel connected to the input port `c` to execute the basic block B2. The CFG is converted into the corresponding actor *FSM* by converting each basic block B1–B16 into a


 Fig. 4. Control Flow Graph (CFG) of the `shuffle` method from Figure 2.

state  $s_1$ – $s_{16}$ , and adding an initial state  $s_0$ . A transition from the initial state  $s_0$  to the state corresponding to the initial basic block, e.g., here  $s_1$  from basic block B1, is added. The action for each transition is the basic block from which the target state of the transition is derived. After compilation of the initial actor *FSM* from the annotated *CFG* post processing rules will be performed on the actor *FSM* to reduce the number of states in the *FSM* and assemble basic blocks into longer actions. The rules will be applied until no more rules match. In order to express these post processing rules more succinctly we define two transitions  $t_a$  and  $t_b$  to be *sequentially mergeable* iff the destination state of transition  $t_a$  is the source state of transition  $t_b$ , i.e.,  $t_a.s' = t_b.s$ <sup>1</sup>, and production of tokens by the action of  $t_a$  implies no consumption of tokens by the action of  $t_b$ , i.e.,  $(\exists o \in O : \text{prod}(t_a, o) > 0) \implies \forall i \in I : \text{cons}(t_b, i) = 0$ . Furthermore, two transitions  $t_a$  and  $t_b$  are *parallel mergeable* iff they have the same source and destination state as well as the same input/output pattern, i.e.,  $t_a.s = t_b.s$ ,  $t_a.s' = t_b.s'$ ,  $\forall i \in I : \text{cons}(t_a, i) = \text{cons}(t_b, i)$ , and  $\forall o \in O : \text{prod}(t_a, o) = \text{prod}(t_b, o)$ . More explicitly, the following post processing rules will be applied to the initial actor *FSM*:

- (1) Elimination of self loops. Self loops which contain no input/output pattern, i.e., the action on the transition of the self loop neither consumes nor produces tokens, can be eliminated by concatenating the contained action with an appropriate `looping construct` for the actions of transitions entering the state with the self loop under elimination.
- (2) Elimination of join states. States with at least one incoming transition and exactly one outgoing transition will be called join states. These states can be eliminated if each incoming transition is sequentially mergeable with the

<sup>1</sup> We use the ‘.’-operator for member access of tuples whose members have been explicitly named in their definition, e.g.,  $t.s$  and  $t.s'$  to denote the current and next state of transition  $t$ .

outgoing transition. The join state is eliminated by concatenating the action of the outgoing transition to the action of the incoming transitions and changing the destination state of the incoming transitions to the destination state of the outgoing transition. Examples of such states are  $s_1, s_2, s_4, s_6, s_7, s_9, s_{12}, s_{13}, s_{14}, s_{15}$ , and  $s_{16}$  from the initial actor *FSM* derived from Figure 4.

- (3) Merging of parallel mergeable transitions. The resulting transition  $t$  from the parallel transitions  $t_a$  and  $t_b$  has the same input/output pattern as  $t_a$  and  $t_b$ , i.e.,  $\forall i \in I : \text{cons}(t, i) = \text{cons}(t_a, i) = \text{cons}(t_b, i) \wedge \forall o \in O : \text{prod}(t, o) = \text{prod}(t_a, o) = \text{prod}(t_b, o)$ , and a guard function consisting of the (possibly simplified) disjunction of the original guard functions, i.e.,  $G_{\text{guard}}(t) = G_{\text{guard}}(t_a) \vee G_{\text{guard}}(t_b)$ . The resulting action of the merged transition will consist of an appropriate **if construct** containing the actions of the original transitions. An example of two compatible parallel transitions result from applying rule (2) to the states  $s_{12}$  and  $s_{13}$  resulting in two compatible transitions between  $s_{11}$  and  $s_{14}$  which can be merged into one transition.

Finally, we do state machine minimization on the resulting transformed action *FSM* and subsequently loop unrolling for loops with known boundaries. Note that these are only loops with loop bodies containing read or write operations, due to the elimination of all other loops by the post processing rules. An example of the actor *FSM* derived by this approach from the code of the `shuffle` method is shown in Figure 5. Note that none of the loops in the `shuffle` actors could be unrolled due to their data dependent iteration count in the `CtrlToken fd`. Therefore, the classification algorithm presented in the next section correctly determines that the `shuffle` actor is neither *SDF* nor *CSDF*.

### 3. 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. A visual representation of all parts of the formal actor description defined below is given in Figure 5. As can be seen, 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.,  $f_{B15;B8}$  or  $g_{B10}$ , 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 3.1 (Actor)** An *actor* is a tuple  $a = (I, O, \mathcal{F}, \mathcal{R})$  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 or evaluation of the general predicates.

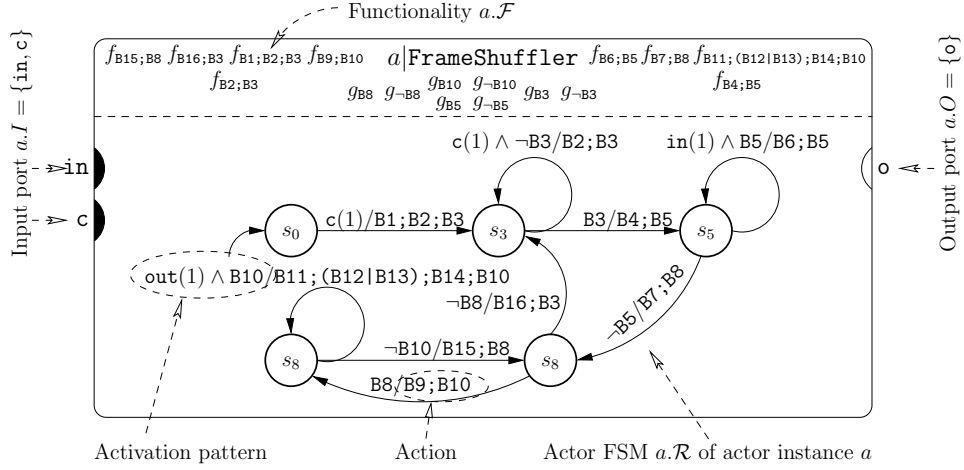


Fig. 5. A visual representation of the formal model of the `FrameShuffler` actor derived from the code shown in the previous section.

As known from data flow descriptions, a data flow graph is a directed graph which models the topology of the actor connections. From an actor's point of view, e.g., actor  $a$  from Figure 5, incoming edges are mapped to the set of *input ports*  $a.I$ , e.g., `in` and `c`, whereas outgoing edges are mapped to the set of *output ports*  $a.O$ , e.g., `o`. These ports are connected to other actor ports via *FIFO* channels. These connections between actor ports are specified by a data flow graph as defined below:

**Definition 3.2 (Data Flow Graph)** A *data flow graph* is a directed graph  $g = (A, C)$  consisting of a set of *actors*  $A$  (vertices) and a set of *channels*  $C \subseteq A.O \times A.I$  (edges).<sup>2</sup>

The basic entity of data transfer are *tokens* transmitted via these channels. Furthermore, there exists a delay function  $D : C \rightarrow \mathbb{N}_0$  which associates with each channel a number of initial tokens, and a *FIFO* capacity function  $S : C \rightarrow \mathbb{N}_0$  specifying the buffer capacity of each channel.<sup>3</sup> The capacity function is constrained to accommodate the initial tokens, i.e.,  $\forall c \in C : D(c) \leq S(c)$ . Figure 1 shows an example for such a data flow graph, with edges and vertices representing *FIFO* channels and *actors*, respectively.

#### 4. CLASSIFICATION

In Section 2, we described how one can separate the communication behavior of an actor from its data path. However, this separation does not yet allow us to apply

<sup>2</sup> We define a set extension of the ‘.’-operator, e.g.,  $A.I = \bigcup_{a \in A} a.I$  and  $A.O = \bigcup_{a \in A} a.O$  to denote the set of all input and output ports of all actors.

<sup>3</sup>As there is a surjective mapping between a port  $p$  and the channel  $c$  it is connected to, we will use  $D(c)$  and  $S(c)$  interchangeably with  $D(p)$  and  $S(p)$ .

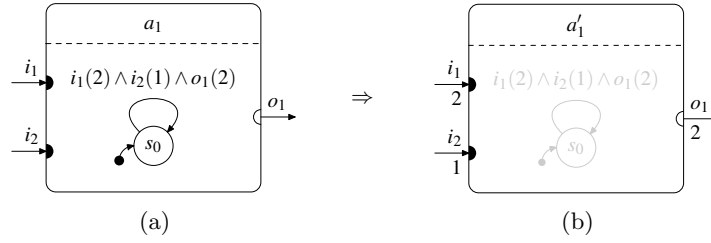


Fig. 6. (a) The possible state transition of actor  $a_1$  consumes two tokens from  $i_1$ , one token from  $i_2$ , and produces two tokens on  $o_1$ . (b) Equivalent *SDF* actor  $a'_1$ .

domain-specific design methods to data flow graphs consisting of actors derived in this way: First we need to recognize important data flow *models of computation* (*MoC*) such as *SDF* and *CSDF*. For this purpose we will briefly reiterate the classification methodology from [Zebelein et al. 2008] which is used to derive the *MoC* of a given actor  $a$  from its *FSM*  $a.\mathcal{R}$ .

The most basic representation of an *SDF* actor in our general data flow notation can be seen in Figure 6(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 by annotating the numbers of tokens consumed and produced in each actor activation to the input and output ports, respectively (cf. 6(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 quasi-static schedule will be calculated by the clustering algorithm presented in Section 5. In this case, an upper bound for the size of each *FIFO* in the subgraph can be determined so that the quasi-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 state of the data path of the actor, only a subset of transitions may be enabled during the execution of the actor. As the state of the data path after the execution of an action is in general unknown, we have to abstract from this state by assuming that a transition  $t$  may always be enabled if its source state  $t.s$  is the current state of the actor during its execution. This abstraction restricts the set of actors with a static communication behavior which will be recognized by the algorithm, i.e., the presented algorithm is only a sufficient criterion.

The classification algorithm presented in [Zebelein et al. 2008] consists of two parts: First, we will show how a specific hypothesis about the communication behavior of the actor — in the following called a *classification candidate* — can be either *accepted* or *rejected*. Subsequently, we will show how to efficiently *construct* these classification candidates.

**Definition 4.1 (Classification candidate)** A classification candidate is a tuple  $c = (\tau, \text{cons}, \text{prod})$ , with  $\tau \geq 1$  being the number of actor phases as known from *CSDF*. To avoid complicated case differentiations, *SDF* actors will simply be treated as *CSDF* actors with  $\tau = 1$ . During the execution of an *CSDF* actor, these actor phases are traversed in a cyclic fashion, i.e., the actor starts in phase 1, transitions to phase 2, etc., until the actor phase wraps around from  $\tau$  to 1. Thus, token consumption and production rates may change in a periodic pattern as specified by the functions *cons* and *prod*: The function  $\text{cons} : \mathbb{N} \times I \rightarrow \mathbb{N}_0$  determines how many tokens are consumed from each input port by the activation of  $a$  in the specified phase. Analogously,  $\text{prod} : \mathbb{N} \times O \rightarrow \mathbb{N}_0$  determines how many tokens are produced on each output port by the activation of  $a$  in the specified phase.

Given a valid classification candidate  $c$ , we now want to check if the states and transitions of the *FSM* of the actor are consistent with  $c$ . If this is the case,  $c$  will be accepted. Otherwise, it will be rejected. For this purpose, a breadth-first search can be initiated in the *FSM* which starts from the initial state  $s_0$  and annotates to each state a *current phase*  $\rho$ , and the number of tokens which remain to be consumed from the actor input ports and produced on the actor output ports in this phase. Then, given an already visited state, each outgoing transition can be evaluated in order to obtain the values for the target states.

In the following, we will use the functions  $\rho : S_{\text{fsm}} \rightarrow \mathbb{N}_0$ ,  $\text{avail} : S_{\text{fsm}} \times I \rightarrow \mathbb{N}_0$  and  $\text{miss} : S_{\text{fsm}} \times O \rightarrow \mathbb{N}_0$  to denote for a given state  $s$  the current actor phase  $\rho(s)$ , the number of tokens  $\text{avail}(s, i)$  yet to be consumed from a given input port  $i$ , and the number of tokens  $\text{miss}(s, o)$  yet to be produced on a given output port  $o$ , respectively. As the actor starts in phase 1, the values annotated to the initial state  $s_0$  can be calculated as follows:  $\rho(s_0) = 1, \forall i \in I : \text{avail}(s_0, i) = c.\text{cons}(1, i)$  and  $\forall o \in O : \text{miss}(s_0, o) = c.\text{prod}(1, o)$ . For all other states, we set  $\rho(s) = 0$  to mark the state as not yet visited.

Given a visited state  $s$  and an outgoing transition  $t = (s, s')$ , the token production and consumption values specified by the transition can be checked against the following conditions:

- (1) If the transition consumes or produces more tokens than indicated by *avail* and *miss*, i.e.,  $\exists i \in I : \text{cons}(t, i) > \text{avail}(s, i)$  or  $\exists o \in O : \text{prod}(t, o) > \text{miss}(s, o)$ , the classification candidate can be immediately rejected, as the transitions belonging to the same actor phase  $\rho(s)$  may not consume and produce more tokens than allotted by  $c$  for this phase. Note that  $c$  is a hypothesis, and if the values for  $c.\text{cons}$  and  $c.\text{prod}$  are chosen too small, this condition will be violated.
- (2) If the transition does not consume all remaining tokens of phase  $\rho(s)$  (i.e.,  $\exists i \in I : \text{cons}(t, i) < \text{avail}(s, i)$ ), it is not allowed to produce any tokens



$c$	$c.\tau$	$c.\text{cons}(1, i)$		$c.\text{prod}(1, o)$		$c.\text{cons}(2, i)$		$c.\text{prod}(2, o)$	
		$i_1$	$i_2$	$o_1$	$o_2$	$i_1$	$i_2$	$o_1$	$o_2$
$c_1$	1	1	1	0	0	-	-	-	-
$c_2$	1	1	2	0	0	-	-	-	-
$c_3$	1	1	2	2	0	-	-	-	-
$c_4$	2	1	2	2	0	1	0	0	0
$c_5$	2	1	2	2	0	2	1	0	1

Table I. Classification candidates constructed from the path defined by the dashed transitions in Figure 7.

In order to systematically construct the classification candidates  $c$  required for the classification algorithm, it can be observed that all paths from  $s_0$  must comply with the assumed parameters  $c.\tau$ ,  $c.\text{cons}$  and  $c.\text{prod}$  if the actor has *SDF* or *CSDF* semantics. Otherwise, the chosen classification candidate  $c$  would be rejected by the search described above. Therefore, one can construct successive classification candidates  $c_i$  by analyzing a *single path*  $p = (t_1, t_2, \dots, t_n)$  from the initial state  $s_0 \in S_{\text{fsm}}$  (i.e.,  $t_1.s = s_0$  and  $t_i.s' = t_{i+1}.s$ ) which returns to a previously visited state. Considering e.g. the *FSM* from Figure 7(a), such a path could be  $p = ((s_0, s_1), (s_1, s_2), (s_2, s_5), (s_5, s_6), (s_6, s_8), (s_8, s_0))$ . Now, for the first candidate  $c_1$ ,  $c_1.\tau = 1$  and the values for  $c_1.\text{cons}(1, i)$  and  $c_1.\text{prod}(1, o)$  are set to the corresponding values of  $t_1$ . If  $c_1$  is rejected, the next candidate  $c_2$  can be constructed by analyzing transition  $t_2$ : Briefly, if tokens are produced by  $t_1$  and consumed by  $t_2$ , a new phase has to be appended to  $c_1$  (i.e.,  $c_2.\tau = 2$ ) because of the possible feedback loops described above, and the values for  $c_2.\text{cons}(2, i)$  and  $c_2.\text{prod}(2, o)$  are set to the corresponding values of  $t_2$ . Otherwise,  $c_2.\tau = 1$ , and the token consumption and production values of  $t_2$  are added to those of  $t_1$  and assigned as values for  $c_2.\text{cons}(1, i)$  and  $c_2.\text{prod}(1, o)$ , i.e. the number of actor phases does not change. In the worst case, a maximum number of  $|S_{\text{fsm}}|$  candidates will be constructed, resulting in an overall complexity of  $O(|S_{\text{fsm}}|(S_{\text{fsm}} + |T|)) = O(|S_{\text{fsm}}|^2 + |S_{\text{fsm}}| |T|)$ . For the *FSM* from Figure 7(a), Table I shows the classification candidates which are constructed from the path defined by the dashed transitions. Candidates  $c_1$  to  $c_4$  are rejected, but  $c_5$  is finally accepted, resulting in the *CSDF* actor shown in Figure 7(b) with two phases and the token consumption and production rates as specified by  $c_5$ .

## 5. CLUSTERING

Now that the static actors of the data flow graph are identified, we want to replace them by a composite actor implementing a *quasi-static schedule* for the static actors. Intuitively, by a quasi-static schedule (*QSS*), we mean a schedule in which a relatively large proportion of scheduling decisions have been made at compile time. Advantages of *QSSs* include lower run-time overhead and improved predictability compared to general dynamic schedules. We will show why composite actors only exhibiting *SDF* or even *CSDF* semantics are insufficient to replace a general static data flow subgraph, i.e., they may introduce deadlocks in the presence of a dynamic data flow environment. Subsequently, we will review our clustering algorithm presented in [Falk et al. 2008]. The clustering algorithm works on vertex-induced

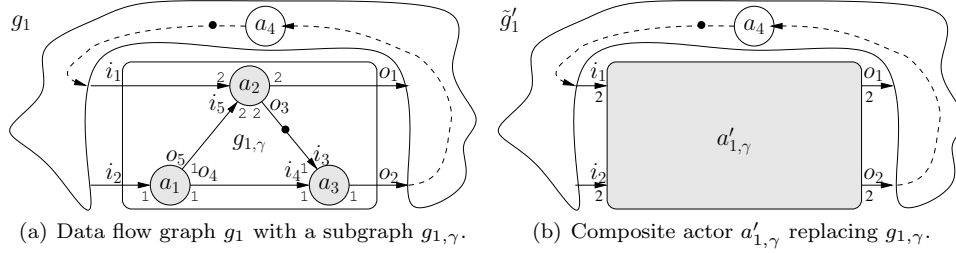


Fig. 8. Example of a dynamic data flow graph  $g_1$  with a subgraph  $g_{1,\gamma}$  and a feedback loop from the subgraph output port  $o_2$  to the subgraph input port  $i_1$  over the actor  $a_4$  and its corresponding conversion into a SDF composite actor  $a'_{1,\gamma}$ .

subgraphs containing only *SDF* and *CSDF* actors, more formally defined below:

**Definition 5.1 (Data Flow Subgraph)** A data flow subgraph  $g_\gamma \subseteq g$  is a vertex-induced subgraph  $g_\gamma = (A, C)$  consisting of a set of actors  $A \subseteq g.A$  (vertices), a set of channels  $C = \{ (o, i) \in g.C \mid \exists a_1, a_2 \in A : o \in a_1.O \wedge i \in a_2.I \}$  (edges).

Analogously to actor ports, we define for a given subgraph  $g_\gamma$  the set of *input ports*  $g_\gamma.I \in A.I$  and the set of *output ports*  $g_\gamma.O \in A.O$ , where  $g_\gamma.I$  and  $g_\gamma.O$  are those actor input and output ports which are connected to channels crossing the subgraph boundary. An example of such an annotated data flow graph is depicted in Figure 8(a) where  $i_1$ ,  $i_2$  and  $o_1$ ,  $o_2$  are the input and output ports of the subgraph  $g_{1,\gamma}$ , respectively. Note that the proposed clustering approach is also able to consider *CSDF* actors. However, for the sake of readability, the examples in this paper use static data flow subgraphs containing only *SDF* actors.

The proposed clustering approach computes for a static data flow subgraph  $g_\gamma$  a *composite actor*  $a_\gamma$  implementing a quasi-static schedule for the actors  $g_\gamma.A$  contained in the original subgraph  $g_\gamma$ . To exemplify the replacement of the subgraph by a single composite actor, we consider again Figure 8(a). The set of static data flow actors  $g_\gamma.A$  consists of the actors  $a_1$ ,  $a_2$  and  $a_3$  which will be mapped — together with some unspecified dynamic data flow actors of  $g_1$  — to a *CPU* of an *MPSoC* system. For this subgraph we have to find a composite actor to replace it. One possibility is shown in Figure 8(b) where the subgraph is replaced by the *SDF* composite actor  $a'_{1,\gamma}$  which implements the static schedule  $(2a_1, a_2, 2a_3)$ , i.e., fire actor  $a_1$  twice then  $a_2$  and finally  $a_3$  twice. The number of firings for each actor is defined by the so called *repetition vector* which can be calculated by solving the *balance equations* [Lee and Messerschmitt 1987]. An *SDF* graph is called *consistent* if the balance equations have a non-trivial solution.<sup>4</sup> The repetition vector assures that after execution of the static schedule sequence the graph returns into its initial state, i.e., the numbers of tokens stored on each edge connecting the actors of the subgraph are identical before and after the execution of the schedule sequence. Furthermore, *CSDF* actors have to be fired an integer multiple of their number of *CSDF* phases. More formally, we define clustering as follows:

<sup>4</sup>The trivial solution being the all zeros repetition vector.

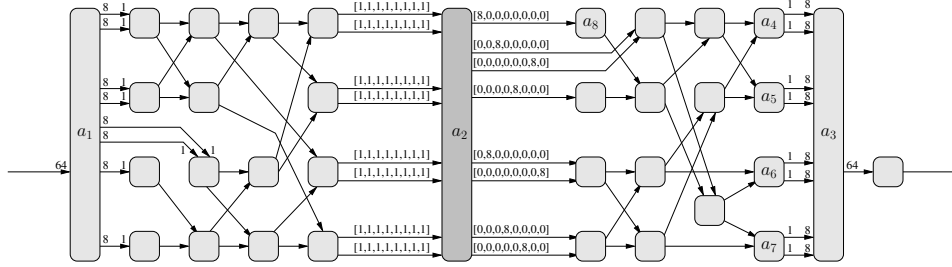


Fig. 9. The above figure depicts the static data flow subgraph contained in the hierarchical IDCT actor from Figure 1. As is customary, the constant consumption and production rates of these *SDF* and *CSDF* actors are annotated on the edges, i.e., a number annotated on the head of an edge corresponds to the consumption rate of the connected actor and a number annotated on the tail corresponds to the production rate of the connected actor. Missing number annotations denote a consumption or production of one token.

**Definition 5.2 (Clustering)** Let  $g$  be a data flow graph and  $g_\gamma \subseteq g$  a static data flow subgraph of  $g$ . Clustering replaces  $g_\gamma$  by a *composite actor*  $a_\gamma$  implementing a quasi-static schedule for the actors  $g_\gamma.A$ , resulting in a new data flow graph  $\tilde{g}$ , i.e.,  $\tilde{g}.A = g.A - g_\gamma.A + \{a_\gamma\}$  and  $\tilde{g}.C = g.C - g_\gamma.C$ .

The goal of our generalized methodology for data flow graph clustering is to provide more flexibility for designers and design tools by trading off among different costs and benefits related to the grouping of subgraphs into individual units for scheduling. The specific clustering algorithm presented in this paper is geared towards reducing dynamic scheduling overhead, while providing enough flexibility to avoid deadlocks during the clustering process, i.e., the data flow schedule implemented by the composite actor will make as few runtime decisions as possible. The clustering algorithm produces a quasi-static schedule in the sense that any scheduling decision which depends on preconditions which cannot be predicted at compile time is still made at runtime. Actor firings which can be executed as a consequence of a precondition are scheduled at compile time resulting in a static schedule for these actor firings.

The more restricted approach which always replaces a static data flow subgraph by its corresponding *SDF* actor implementing a static schedule for the actors of the subgraph excels at schedule overhead reduction.<sup>5</sup> An example of such a best case scenario is shown in Figure 9 where the *SDF* subgraph for the 2-dimensional IDCT can be replaced by an *SDF* composite actor implementing a static schedule of the contained IDCT actors, i.e., for the composite actor replacing the IDCT subgraph we only need to check the prerequisite of 64 tokens on the input of the IDCT before we can apply the calculated static schedule. Indeed, this is a special case of our clustering methodology which also computes a fully static schedule for the IDCT subgraph.

<sup>5</sup>In general, the longer the static scheduling sequence which can be executed by checking a single prerequisite, the less schedule overhead is imposed by this schedule.

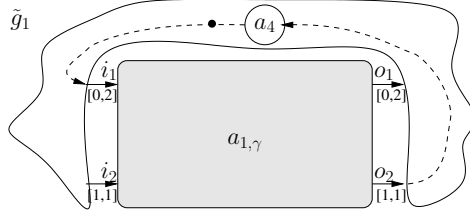


Fig. 10. Instead of converting the subgraph  $g_{1,\gamma}$  into an *SDF* actor, an alternative conversion into the above depicted *CSDF* actor solves the introduction of the deadlock from Figure 8(b).

However, this approach might be infeasible in the general case due to the *environment* of the static data flow subgraph. An example for such a problematic environment is depicted as a cloud in Figure 8(a). The problem stems from the lack of flexibility in the static schedule  $(2a_1, a_2, 2a_3)$  implemented by the composite *SDF* actor  $a'_{1,\gamma}$  shown in Figure 8(b). The feedback loop from the output port  $o_2$  to the input port  $i_1$  (represented by the dashed edges and the dynamic data flow actor  $a_4$ ) adds a data dependency which implies that actor  $a_3$  has to be executed at least once before executing actor  $a_2$  (Assuming actor  $a_4$  will not spontaneously produce tokens). However, this dependency cannot be satisfied due to the constrained sequence of actor firings by the static schedule  $(2a_1, a_2, 2a_3)$  which fires actor  $a_2$  before firing actor  $a_3$ . The problem can be solved by converting the subgraph  $g_{1,\gamma}$  into a *CSDF* composite actor  $a_{1,\gamma}$  shown in Figure 10 which has two phases and thus implements a quasi-static schedule. In the first phase a positive check for one token on port  $i_2$  is followed by the execution of the static schedule  $(a_1, a_3)$ , whereas in the second phase a positive check for two tokens on port  $i_1$  and one token on port  $i_2$  results in the execution of the static schedule  $(a_1, a_2, a_3)$ .

However, this strategy fails when applied to the next example shown in Figure 11(a). In this case we assume a dynamic actor  $a_4$  which uses two different forwarding modes. In mode (i), it forwards both depicted tokens to port  $i_1$ . This allows production of two tokens on port  $o_1$  which are then forwarded to port  $i_2$  by actor  $a_4$ . In mode (ii) however, actor  $a_4$  starts to forward one token to port  $i_2$ . This time, the subgraph  $g_{1,\gamma}$  can produce one token on port  $o_2$  which allows actor  $a_4$  to forward two tokens to port  $i_1$ . After generation of two tokens on output port  $o_1$ , actor  $a_4$  finally forwards again one token to port  $i_2$ .

Unfortunately we can recognize that applying the above schedule,  $(a_1, a_3)$  followed by  $(a_1, a_2, a_3)$ , fails if actor  $a_4$  forwards tokens according to forwarding mode (i). We could try to solve this problem by substituting another *CSDF* composite actor  $a'_{2,\gamma}$  shown in Figure 11(b) which checks for two tokens on  $i_1$ , executes schedule  $(a_2)$ , checks for two tokens on  $i_2$  and finally executes schedule  $(2a_1, 2a_3)$ . However, this schedule fails for forwarding mode (ii).

More explicitly, forwarding mode (i) of actor  $a_4$  requires firing actor  $a_2$  first while forwarding mode (ii) implies firing  $(a_1, a_3)$  first. As the dynamic actor can arbitrarily switch between the two modes, the subgraph  $g_{2,\gamma}$  can neither be converted into an *SDF* nor a *CSDF* actor without possibly introducing a deadlock. Therefore, the decision which schedule sequence to choose depends on token availability

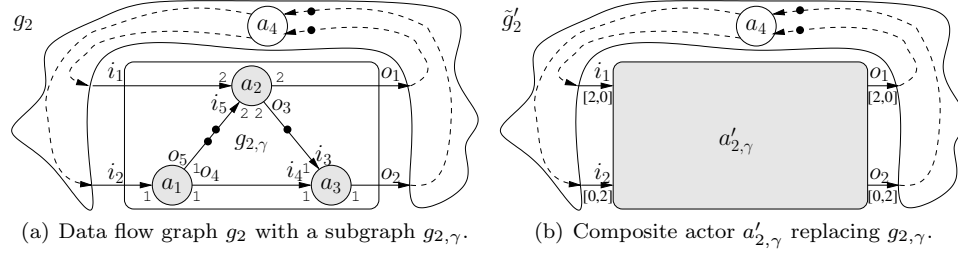


Fig. 11. Second example with two feedback loops  $o_1 \rightarrow i_2$  and  $o_2 \rightarrow i_1$  over the dynamic data flow actor  $a_4$  and a CSDF composite actor  $a'_{2,\gamma}$  replacement candidate for subgraph  $g_{2,\gamma}$ . The dynamic data flow actor  $a_4$  randomly switches between forwarding mode (i) and forwarding mode (ii). Where forwarding mode (i) forwards two tokens to port  $i_1$  followed by two tokens to port  $i_2$  and forwarding mode (ii) forwards one token to port  $i_2$ , two tokens to port  $i_1$  followed finally by one token to port  $i_2$ .

on the subgraph input ports  $i_1$  and  $i_2$ . However, this flexibility cannot be implemented by a static schedule. Instead, a quasi-static schedule (*QSS*) is required, which statically schedules the sequences  $(a_1, a_3)$  and  $a_2$  but postpones the decision which sequence to execute first to runtime.

The above examples demonstrate that in the general case a static data flow subgraph cannot be converted into an *SDF* nor *CSDF* composite actor without possibly introducing a deadlock into the transformed data flow graph. In order to represent a quasi-static schedule computed by our clustering algorithm, we introduce the so called *cluster finite state machine (FSM)*:

**Definition 5.3 (Cluster FSM)** The *cluster FSM* is a tuple  $\mathcal{R} = (S_{\text{fsm}}, s_0, T, \text{cons}, \text{prod}, F_{\text{sched}})$  containing a set of *states*  $S_{\text{fsm}}$ , an *initial state*  $s_0 \in S_{\text{fsm}}$ , a set of *transitions*  $(s, s') \in T \subseteq S_{\text{fsm}} \times S_{\text{fsm}}$  containing the current state  $s$  and the next state  $s'$ , a function  $\text{cons} : T \times I \rightarrow \mathbb{N}_0$  denoting the number of tokens  $\text{cons}(t, i)$  which must be available on input port  $i$  for the execution of transition  $t$ , a function  $\text{prod} : T \times O \rightarrow \mathbb{N}_0$  denoting the amount of free token places  $\text{prod}(t, o)$  which must be available on output port  $o$  for the execution of transition  $t$ , and a action mapping function  $F_{\text{sched}} : T \rightarrow \mathcal{F}$  mapping a transition  $t$  to a scheduling action  $f_{\text{sched}} = F_{\text{sched}}(t) \in g_\gamma \cdot A^*$  for the contained actors.

As can be seen from the previous definition, a cluster *FSM* is an actor *FSM* which simply has another sort of functionality  $\mathcal{F}$ . Instead of operations for the data path it contains a scheduling sequence for the contained actors in the cluster  $g_\gamma$ , i.e.,  $\mathcal{F} \subseteq g_\gamma \cdot A^*$ .

With this notation, the subgraph  $g_{2,\gamma}$  can be replaced by a composite actor  $a_{2,\gamma}$  which can be expressed as shown in Figure 12(b). Note that we use  $p(n)$  to denote that at least  $n$  tokens/free space must be available on the channel connected to the actor port  $p$ . As can be seen, two transitions  $t_1$  and  $t_2$  are leaving the start state  $s_0$ . Whereas  $t_1$  requires at least two tokens on input port  $i_1$  (denoted by the precondition  $i_1(2)$ ) and executes the static schedule  $(a_2)$ ,  $t_2$  requires at least one input token on input port  $i_2$  (denoted by  $i_2(1)$ ) and executes the static schedule  $(a_1, a_3)$ . This

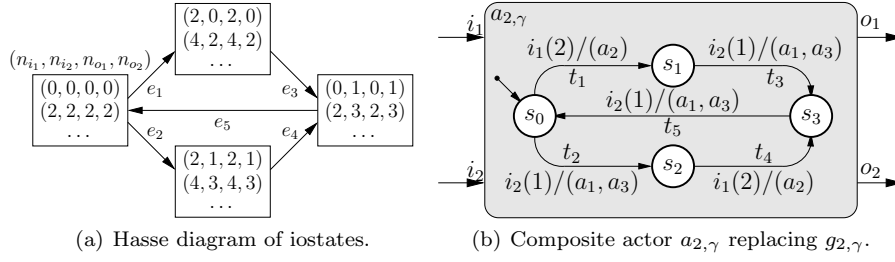


Fig. 12. The composite actor  $a_{2,\gamma}$  replacing subgraph  $g_{2,\gamma}$  can be represented via a *cluster Finite State Machine*. The cluster *FSM* is derived from the Hasse diagram of the iostates of the  $g_{2,\gamma}$  subgraph. The transitions are annotated with preconditions and the static scheduling sequences to execute if a transition is taken.

notation allows us to represent static schedules (the *FSM* has one state with one self-transition), quasi-static schedules (the *FSM* has at least one transition with a scheduling sequence of more than one actor), and dynamic schedules (all transitions of the *FSM* have a degenerated scheduling sequence containing exactly one actor). However, the proposed clustering algorithm requires the following prerequisite:

**Definition 5.4 (Clustering Condition)** A static data flow subgraph  $g_\gamma$  can be clustered by the given algorithm if the subgraph disregarding its inputs and outputs is deadlock free itself and for each pair of cluster input/output ports  $(i, o)$  there exists a directed path  $p$  from  $i$  to  $o$  in the subgraph  $g_\gamma$ , i.e.,  $\forall (i, o) \in g_\gamma.I \times g_\gamma.O : \exists p = ((o_1, i_1), \dots, (o_n, i_n)) \in g_\gamma.C^*$  such that  $\exists a \in g_\gamma.A : i \in a.I \wedge o_1 \in a.O$  and  $\exists a \in g_\gamma.A : i_n \in a.I \wedge o_n \in a.O$  and  $\forall k, 1 \leq k < n : \exists a \in g_\gamma.A : i_k \in a.I \wedge o_{k+1} \in a.O$ .

Otherwise an unbounded accumulation of tokens will result. To exemplify this consider the subgraph  $g_{2,\gamma}$  depicted in Figure 11(a) satisfying the clustering condition. However, removing the channel  $(o_5, i_5)$  would contradict the condition as there is no path from  $i_2$  to  $o_1$ , thus leading to an unbounded accumulation of tokens on edge  $(o_3, i_3)$  because producing tokens on port  $o_1$  would never require consuming any tokens from port  $i_2$ . The unbounded accumulation would lead to an unbounded number of states of the cluster *FSM* as the state space of the fill levels of the channels  $g_\gamma.C$  is represented in the state space of the cluster *FSM*. For static data flow subgraphs that do not satisfy the cluster condition, the set of static actors  $g_\gamma.A$  can be partitioned into subsets by removing the channels on which unbounded accumulation of tokens can occur thus splitting the subgraph into connected subgraphs satisfying the *clustering condition*.

The key idea for preserving deadlock freedom is to satisfy the worst case environment of the subgraph. This environment contains for each output port  $o \in g_\gamma.O$  and each input port  $i \in g_\gamma.I$  a feedback loop where any produced token on the output port  $o$  is needed for the activation of an actor  $a \in g_\gamma.A$  connected to input port  $i$ . In particular, postponing the production of an output token results in a deadlock of the entire system. Hence, the *QSS* determined by our clustering algorithm guarantees the production of a maximum number of output tokens from the consumption of a minimal number of input tokens.

$(n_{i_1}, n_{i_2},$	(0, 0,	(0, 1,	(2, 0,	(2, 1,	(2, 2,	(2, 3,	(4, 2,	(4, 3,	(4, 4,	...
$n_{o_1}, n_{o_2})$	0, 0)	0, 1)	2, 0)	2, 1)	2, 2)	2, 3)	4, 2)	4, 3)	4, 4)	...

Table II. Minimal number of input token  $n_{i_1}/n_{i_2}$  necessary on subgraph  $g_{2,\gamma}$  input port  $i_1/i_2$  and maximal number of output tokens  $n_{o_1}/n_{o_2}$  producible on the subgraph output ports  $o_1/o_2$  with this input tokens. For example, to produce two token on output port  $o_2$ , two tokens are required from both input ports  $i_1$  and  $i_2$ . The maximal number of output tokens producible from these input tokens are two output tokens on  $o_1$  and  $o_2$ .

For the subgraph  $g_{2,\gamma}$  this relation is given in Table II which depicts the *iostates* of the subgraph  $g_{2,\gamma}$ . For an efficient computation of this table we refer the interested reader to [Falk et al. 2008].

We first note that this table is infinite but periodic as can be seen from the depiction of the *iostates* in the corresponding Hasse diagram (cf. 12(a)). Equivalent values, i.e., *iostates* which can be reached by firing all actors of the subgraph according to the *repetition vector* of the subgraph, are aggregated into one vertex in the Hasse diagram. This is also the reason why the Hasse diagram contains cycles as normally these values would all be depicted as unique vertices. As can be seen from Figure 12(a) and Figure 12(b) there is a one-to-one correspondence between the vertices and edges in the Hasse diagram and the states and transition in the cluster FSM. For a more technical explanation of deriving the cluster FSM from the Hasse diagram we again refer the reader to [Falk et al. 2008].

## 6. I/O BUFFER SIZE ADJUSTMENT

The clustering algorithm presented in the previous section calculates a *QSS* for the actors contained in the cluster. During the execution of the application, the cluster *FSM* only needs to check if there are enough tokens on the cluster input ports and free spaces on the cluster output ports in order to execute the static scheduling sequence annotated to a transition of the cluster *FSM*. The buffer requirements that are needed *internally* by the cluster in order to produce some tokens on the cluster output ports after having consumed some tokens from the cluster input ports can be easily determined from the cluster *FSM* during the synthesis of the application. However, the sizes of the *FIFOs* connected to the cluster input and output ports as specified by the developer of the application may neither guarantee that (i) all transitions of the cluster *FSM* could potentially be enabled, nor that (ii) the clustered network graph can be executed without deadlocks.

Problem (i) stems from the fact that multiple firings of the same actor may be specified by a static scheduling sequence. Considering the subgraph  $g_1$  shown in Figure 13(a), the user could have specified the following *FIFO* sizes:  $S(c_1) = 1$ ,  $S(c_2) = 1$  and  $S(c_3) = 5$ . Now, assume that we want to execute  $a_1$  five times and  $a_2$  once. Obviously, in the dynamically scheduled data flow graph, the *FIFO* sizes as specified by the user are sufficient for achieving this goal: Wait for a token on  $c_1$ , fire  $a_1$  once, wait for another token on  $c_1$ , fire  $a_1$  for the second time, etc., until  $a_1$  has been activated five times. Subsequently, we have to wait for a free buffer place on channel  $c_2$  before we can finally execute  $a_2$ .

On the other hand, after  $g_1$  has been clustered into the composite actor  $\gamma_1$  (shown in Figure 13(b)), the only transition of the cluster *FSM* requires that we must wait

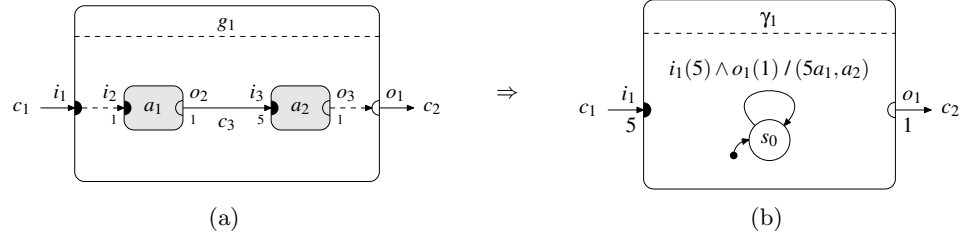


Fig. 13. Data flow subgraph  $g_1$  (a) and the corresponding composite SDF actor  $\gamma_1$  (b).

for five tokens on  $i_1$  and one free buffer place on  $o_1$  before the transition can be executed. However, the channel size of  $c_1$  is still 1, which means that the transition can never be enabled, thus (in this case) deadlocking the model.

Problem (ii) stems from the fact that some buffer capacities in the internal buffers of the original data flow subgraph are lost by the clustering operation. Considering again the subgraph  $g_1$  shown in Figure 13(a), the user could also have specified the following *FIFO* sizes:  $S(c_1) = 5$ ,  $S(c_2) = 1$  and  $S(c_3) = 10$ . Obviously, in the dynamically scheduled data flow graph,  $a_1$  can be activated *independently* from  $a_2$ , i.e., assuming no firings of actor  $a_2$  a source actor  $a_0$  connected to channel  $c_1$  can produce up to 15 tokens (assuming  $c_1$  and  $c_3$  were initially empty) by ten intermittent firings of actor  $a_1$  (thus filling channel  $c_3$ ) leaving five free buffer places on channel  $c_1$  which can be filled subsequently by  $a_0$ .

On the other hand, after  $g_1$  has been clustered into the composite actor  $\gamma_1$  (shown in Figure 13(b)), the scheduling sequence  $(5a_1, a_2)$  always consumes five tokens from the cluster input port  $i_1$ , and atomically produces one token on the cluster output port  $o_1$ . If the dynamic environment demands that a total of 10 tokens must be produced on  $c_1$  before a token can be consumed from  $c_2$ , the composite actor  $\gamma_1$  will deadlock the model, as in this case only up to five tokens can be produced on  $c_1$  before a token will be produced on  $c_2$ . This means that the buffer capacity of  $c_3$  is lost due to the clustering operation.

It is theoretically possible to solve this problem by modeling each bounded buffer with two unbounded ones and using the transformed model as input for the clustering algorithm. However, as the state space for the resulting cluster *FSM* would significantly increase, this model transformation would in turn significantly degrade the performance improvements that can be achieved with the clustering operation.

In this section, we present an algorithm to calculate an upper bound for the sizes of the input and output buffers connected to the composite actor such that problem (i) is satisfied, and the buffer capacity increment accommodates the memory which is lost by the clustering operation due to problem (ii).

As can be seen in Figure 12(b) on Page 18, the cluster *FSM* may contain more than one state. As the dynamic environment is not aware of the current state  $s_{\text{cur}}$  of a cluster *FSM*, a deadlock free execution of the data flow graph must be guaranteed for each state of the cluster *FSM*. Thus, the following steps can be identified:

- (1) For each state of the cluster *FSM*, the internal buffer capacities lost due to the clustering operation are determined. It will be seen that this hidden memory may be *redistributed* from the cluster input channels to the cluster output

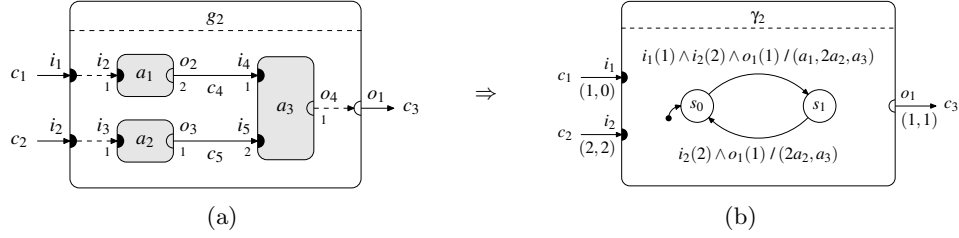


Fig. 14. Data flow subgraph  $g_2$  with  $FIFO$  sizes  $S(c_1) = S(c_2) = S(c_3) = 1$  and  $S(c_4) = S(c_5) = 4$ , and (b) the corresponding composite  $CSDF$  actor  $\gamma_2$  (b).

channels according to the cluster  $FSM$ . For each such memory redistribution scheme, one can calculate buffer sizes for the cluster input and output channels satisfying the requirements outlined above.

- (2) The buffer sizes calculated in step (1) define a search space which is explored in this step, i.e., a memory redistribution scheme has to be selected from each state. The presented algorithm allows for the exact solution of this selection problem. In general, however, it will be seen that the search space may become very large, which is why we will use a heuristic approach in this paper.

### 6.1 Inaccessible Memory

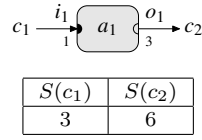
In order to calculate the inaccessible memory contained in the cluster, one can first determine the number of initial tokens on the  $FIFO$  channels of the cluster for each state  $s \in S_{\text{fsm}}$ . In the following, the function  $D_{\text{state}} : S_{\text{fsm}} \times C \rightarrow \mathbb{N}_0$  will be used to encode these token numbers. The values of  $D_{\text{state}}$  can be calculated with the help of the static scheduling sequences annotated to the transitions of the cluster  $FSM$ , under the starting condition  $\forall c \in C : D_{\text{state}}(s_0, c) = D(c)$ .

Based on  $D_{\text{state}}$ , the maximum number of tokens that can be consumed from the cluster input ports without producing any tokens on the cluster output ports can be calculated. As for each state of the cluster  $FSM$  a different number of delays may be contained in the internal  $FIFOs$ , these token numbers may also differ between the states of the cluster  $FSM$ . In the following, the function  $\text{cons}_{\text{max}} : S_{\text{fsm}} \times I \rightarrow \mathbb{N}_0$  will be used to encode these token numbers.

Considering the subgraph  $g_2$  from Figure 14(a), the user did not specify any delays for both channels internal to the subgraph, i.e.,  $D(c_4) = D(c_5) = 0$ . Therefore, the following values can be calculated for  $D_{\text{state}}$  and  $\text{cons}_{\text{max}}$  based on the corresponding cluster  $FSM$  shown in Figure 14(b):

$s$	$D_{\text{state}}(s, c)$		$\text{cons}_{\text{max}}(s, i)$	
	$c_4$	$c_5$	$i_1$	$i_2$
$s_0$	0	0	2	4
$s_1$	1	0	1	4

It should be noted that during transition  $(s_0, s_1)$ ,  $a_1$  produces two tokens on  $c_4$ , but  $a_3$  only consumes one token from  $c_4$ , leaving one token on  $c_4$  (i.e.,  $D_{\text{state}}(s_1, c_4) = 1$ ). As a result,  $a_1$  could only be fired once in  $s_1$  (due to the remaining token on  $c_4$  in this state) before  $a_3$  would have to be activated (i.e.,  $\text{cons}_{\text{max}}(s_1, i_1) = 1$ ).

(a) *SDF* actor

$S_{\text{adj}}(c_1)$	$S_{\text{adj}}(c_2)$
4	3
3	6
2	9
1	12

(b) Valid buffer sizes

Fig. 15. If the number of tokens consumed and produced per actor firing is known (a), memory in a network graph can be redistributed accordingly ( $S_{\text{adj}}$  denotes the adjusted *FIFO* sizes) (b).

Obviously,  $\text{cons}_{\text{max}}$  represents the inaccessible memory contained in the cluster which has to be made available again outside the cluster in order to avoid the introduction of deadlocks into the clustered data flow graph. Subsequently, this inaccessible memory will be redistributed between the cluster input and output channels in order to satisfy the buffer size requirements imposed by the cluster *FSM*.

The first buffer size requirement was outlined as problem (i) on page 19 and stems from the fact that multiple firings of the same actor may be specified by a static scheduling sequence. Therefore, for each actor connected to some cluster input or output ports, the corresponding channels must be sufficiently large in order to accommodate the accumulated port requirements for these multiple activations. As the transitions of the cluster *FSM* may specify different requirements for the same cluster input or output port, we have to calculate the maximum over each transition.

The second buffer size requirement simply states that we are not allowed to *reduce* the sizes of the channels connected to the cluster input and output ports, i.e., we must respect the user-defined *FIFO* sizes. This stems from the fact that reducing buffer sizes may block transitions of actors not contained in the cluster but connected to the cluster input and output channels.

In the following, the functions  $S_{\text{min}} : I \cup O \rightarrow \mathbb{N}_0$  will be used to encode these minimum size requirements, i.e.,  $\forall i \in I : S_{\text{min}}(i) = \max\{S(i), \max\{\text{cons}(t, i) \mid t \in T\}\}$  and  $\forall o \in O : S_{\text{min}}(o) = \max\{S(o), \max\{\text{prod}(t, o) \mid t \in T\}\}$ .

For the example from Figure 14,  $S_{\text{min}}(i_1) = 1$ ,  $S_{\text{min}}(i_2) = 2$  and  $S_{\text{min}}(o_1) = 1$ . The next step is concerned with the calculation of possible memory redistribution schemes such that  $S_{\text{min}}$  is satisfied.

## 6.2 Memory Redistribution

Figure 15 shows how this memory redistribution can be accomplished for simple *SDF* actors: The actor parameters specify how much the size of the channels connected to the input ports must be decreased, and the size of the channels connected to the output ports increased (or vice versa). This can be carried out iteratively until one of the channels has reached its minimum size. Generally, however, the cluster *FSM* does not exhibit *SDF* semantics, i.e., there may be several transitions which consume and produce different numbers of tokens, resulting in a more complicated memory redistribution procedure, which will be explained in this section. Starting from the *current state*  $s_{\text{cur}} \in S_{\text{fsm}}$ , each path defines how input tokens may be transformed into output tokens. Consider the example from Figure 14(b):

		cons <sub>acc</sub> / prod <sub>acc</sub>		
s <sub>cur</sub>	s	i <sub>1</sub>	i <sub>2</sub>	o <sub>1</sub>
s <sub>0</sub>	s <sub>0</sub>	0	0	0
	s <sub>1</sub>	1	2	1
s <sub>1</sub>	s <sub>0</sub>	0	2	1
	s <sub>1</sub>	0	0	0

Table III. Accumulated port requirements for the example from Figure 14

If  $s_0$  is the current state, transition  $(s_0, s_1)$  says that we have to decrease the size of  $c_1$  by 1,  $c_2$  by 2 and increase the size of  $c_3$  by 1. On the other hand, if  $s_1$  is the current state, we have to consider transition  $(s_1, s_0)$ , i.e., we have to decrease the size of  $c_2$  by 2 and increase the size of  $c_3$  by 1 (Note that the size of  $c_1$  can not be decreased).

These *accumulated port requirements* will in the following be encoded by the functions  $\text{cons}_{\text{acc}} : S_{\text{fsm}} \times S_{\text{fsm}} \times I \rightarrow \mathbb{N}_0$  and  $\text{prod}_{\text{acc}} : S_{\text{fsm}} \times S_{\text{fsm}} \times O \rightarrow \mathbb{N}_0$  which map the current state  $s_{\text{cur}}$ , an arbitrary state  $s$  reachable from  $s_{\text{cur}}$  and a cluster input port  $i$  or cluster output port  $o$  to the accumulated port requirements (i.e., tokens or free spaces) for this port. The values for both functions can be determined by a search in the *FSM*, under the starting condition  $\forall i \in I : \text{cons}_{\text{acc}}(s_{\text{cur}}, s_{\text{cur}}, i) = 0$  and  $\forall o \in O : \text{prod}_{\text{acc}}(s_{\text{cur}}, s_{\text{cur}}, o) = 0$ . Table III shows  $\text{cons}_{\text{acc}}$  and  $\text{prod}_{\text{acc}}$  for the example from Figure 14.

It should be noted that the values of  $\text{cons}_{\text{acc}}$  and  $\text{prod}_{\text{acc}}$  only specify the accumulated port requirements for a *single* execution of the repetition vector of the cluster. In the general case, however, the port requirements of the cluster *FSM* may only be satisfied if a state  $s$  is visited more than once. This can only happen if  $s$  is part of a cyclic path, i.e.,  $s$  must be contained in the strongly connected component of the cluster *FSM*. Due to the way the cluster *FSM* is constructed, the accumulated port requirements over any such cyclic path evaluates exactly to the so called *input/output repetition vector*, which is defined as the number of tokens required on the cluster input ports and free spaces required on the cluster output ports in order to execute the whole repetition vector of the cluster. In the following, we will use the functions  $\text{cons}_{\text{rep}} : I \rightarrow \mathbb{N}$  and  $\text{prod}_{\text{rep}} : O \rightarrow \mathbb{N}$  to encode these token numbers. For the subgraph shown in Figure 14,  $\text{cons}_{\text{rep}}(i_1) = 1$ ,  $\text{cons}_{\text{rep}}(i_2) = 4$ , and  $\text{prod}_{\text{rep}}(o_1) = 2$ .

Putting it all together, we can define two functions  $\text{dec} : S_{\text{fsm}} \times S_{\text{fsm}} \times \mathbb{N}_0 \times I \rightarrow \mathbb{N}_0$  such that  $\text{dec}(s_{\text{cur}}, s, k, i) = \text{cons}_{\text{acc}}(s_{\text{cur}}, s, i) + k \cdot \text{cons}_{\text{rep}}(i)$  and  $\text{inc} : S_{\text{fsm}} \times S_{\text{fsm}} \times \mathbb{N}_0 \times O \rightarrow \mathbb{N}_0$  such that  $\text{inc}(s_{\text{cur}}, s, k, o) = \text{prod}_{\text{acc}}(s_{\text{cur}}, s, o) + k \cdot \text{prod}_{\text{rep}}(o)$ .

In the following, the tuple  $(s_{\text{cur}}, s, k)$  always consists of a given current state  $s_{\text{cur}}$ , an arbitrary state  $s$  reachable from  $s_{\text{cur}}$ , and a repetition number  $k \geq 0$ . Given such a tuple, we can decrease the size of the channel connected to each cluster input port  $i$  by  $\text{dec}(s_{\text{cur}}, s, k, i)$  while simultaneously increasing the size of the channel connected to each cluster output port  $o$  by  $\text{inc}(s_{\text{cur}}, s, k, o)$ . If  $s$  is not part of a strongly connected component, it can not be visited more than once, hence, we have to set  $k = 0$  in this case.

Now, for a given tuple  $(s_{\text{cur}}, s, k)$ , the corresponding adjusted *FIFO* sizes can be calculated as follows:  $\forall i \in I : S_{\text{adj}}(i) = \max\{S_{\text{min}}(i), S(i) + \text{cons}_{\text{max}}(s_{\text{cur}}, i) -$

$s_{\text{cur}}$	$p$	$(s, k)$				
		$(s_0, 0)$	$(s_0, 1)$	$(s_0, 2)$	$(s_1, 0)$	$(s_1, 1)$
$s_0$	$i_1$	<b>3</b>	<b>2</b>	1	<b>2</b>	<b>1</b>
	$i_2$	<b>5</b>	<b>2</b>	2	<b>3</b>	<b>2</b>
	$o_1$	<b>1</b>	<b>3</b>	5	<b>2</b>	<b>4</b>
$s_1$	$i_1$	<b>2</b>	1	-	<b>2</b>	<b>1</b>
	$i_2$	<b>3</b>	2	-	<b>5</b>	<b>2</b>
	$o_1$	<b>2</b>	4	-	<b>1</b>	<b>3</b>

Table IV. Possible sets of adjusted buffer sizes for the example from Figure 14

$\text{dec}(s_{\text{cur}}, s, k, i)$  and  $\forall o \in O : S_{\text{adj}}(o) = \max\{S_{\text{min}}(o), S(o) + \text{inc}(s_{\text{cur}}, s, k, o)\}$ , i.e., we add the hidden memory  $\text{cons}_{\text{max}}$  to the cluster input channels, and then redistribute some memory from the cluster input channels to the cluster output channels according to  $\text{dec}$  and  $\text{inc}$ .

It should be noted that the maximum operation is needed in order to satisfy the buffer size requirements imposed by  $S_{\text{min}}$ , which could otherwise be violated (e.g., if we did redistribute too much memory). As we assume that adding buffer capacities to the input and output channels does not introduce deadlocks into the data flow graph, this is a valid operation.

For the example from Figure 14, the sets of adjusted buffer sizes are given in Table IV. Two observations can be made: For each tuple  $(s_{\text{cur}}, s)$  there exists an upper bound for the repetition number  $k$  up to which the adjusted buffer sizes need to be calculated. This stems from the fact that during the calculation of  $S_{\text{adj}}$  a maximum operation is applied to enforce the minimum buffer sizes  $S_{\text{min}}$  (see above).

Furthermore, for a given current state  $s_{\text{cur}}$ , a set of adjusted buffer sizes *dominates* another set iff for each port the value of  $S_{\text{adj}}$  from the first set is smaller than or equal to the corresponding value of the second set. In the table given above, all sets of  $S_{\text{adj}}$  which are not dominated by other sets of  $S_{\text{adj}}$  are highlighted. Obviously, dominated sets can be removed from the search space, because they will not produce better solutions than the sets which are not dominated.

### 6.3 Search Space Exploration

The last step is the selection of a non-dominated set of adjusted buffer sizes for each state. The adjusted buffer size for the channel connected to each cluster input and output port will then be set to the *maximum* over the corresponding value of  $S_{\text{adj}}$  from each selected set. For the example from Figure 14, if we select the sets corresponding to the tuples  $(s_0, s_1, 1)$  and  $(s_1, s_0, 0)$  (cf. Table IV), the resulting *FIFO* sizes will be  $S(c_1) = \max\{1, 2\} = 2$ ,  $S(c_2) = \max\{2, 3\} = 3$  and  $S(c_3) = \max\{4, 2\} = 4$ .

A sensible target function of this selection process can e.g. be defined as “minimize the total number of bytes added to the cluster input and output channels”. However, as in the general case an exhaustive search is prohibitive, and this optimization problem is out of scope for this paper, we use a rather simple heuristic which determines for each state the set of buffer redistribution values which locally minimizes the number of bytes added to the cluster input and output channels re-

sulting in valid but not necessarily optimal sizes for the input and output channels of the cluster.

For the example from Figure 14, the sets corresponding to the tuples  $(s_0, s_1, 1)$  and  $(s_1, s_1, 1)$  (cf. Table IV) are selected, resulting in the *FIFO* sizes  $S(c_1) = 1$ ,  $S(c_2) = 2$  and  $S(c_3) = 4$ .

## 7. RESULTS

In order to evaluate the optimization potential for real-world examples, we have applied our clustering algorithm to the Motion-*JPEG* decoder. The clustering methodology clusters the *IDCT* subgraph into an *SDF* composite actor. The total runtime for 100 *QCIF* images (176x144) has been measured on a Intel Pentium 4 CPU with 3.00GHz for a fully dynamic scheduled (6.8 seconds) and a dynamic schedule with the *SDF* composite actor (3.1 seconds).

Furthermore, we have measured the speedup (factor 4.5) for only the two-dimensional *IDCT* subgraph of our Motion-*JPEG* decoder. The runtime was measured for 100,000  $8 \times 8$ -blocks of *IDCT* data. In the unclustered case 2.1 seconds was measured while the clustered case achieved a total runtime of 0.47 seconds.

Both measurements have been taken on a Pentium 4 CPU using only a single thread of execution (single processor schedule). To evaluate the methodology for embedded systems we have applied our clustering algorithm to the two-dimensional *IDCT* of our Motion-*JPEG* decoder for both a single-processor and a multi-processor implementation. Both of them were implemented using MicroBlaze processors running at 66 MHz in a Xilinx Virtex-II Pro FPGA. For the single-processor implementation we measured the total runtime for 200  $8 \times 8$ -blocks of *IDCT* data. We achieved a speedup of factor 2 with an execution time of 1.91 seconds for the fully dynamic scheduling strategy whereas the *QSS* computed by our clustering approach only takes 0.95 seconds. The different speedup factors between the PC and the MicroBlaze is explained by the different optimizations in the code generation phase of our clustering tools. The measurements and code for the MicroBlaze platform were generated with an older code base of the clustering tools which lacked optimizations pertaining to the reduction of *FIFO* channels of size one to registers whereas the PC measures are of newer origin benefiting from these optimizations. Indeed, we have measured the clustered *IDCT* without the *FIFO* to register reduction on the PC and obtained a runtime of 1.1 seconds which results in a speedup of factor 1.9.

For the multi-processor implementation, we partitioned the *IDCT* into four clusters as shown in Figure 16. Each of the four composite actors resulting from applying our clustering algorithm is implemented on a single MicroBlaze. Inter-processor communication is implemented using Xilinx Fast Simplex Links (*FSL*). The source (s) and the sink (d) are implemented as hardware modules. Comparing the dynamic schedule with the *QSS* resulting from the clustering algorithm, the latter improves the throughput by 45% (dynamic: 1259 blocks/s, *QSS*: 1831 blocks/s).<sup>6</sup>

<sup>6</sup>The super-linear speedup in comparison to the single processor implementation is due to the different memory interfaces used in both implementations: Thanks to the multi-processor implementation, the code size for the individual processors could be reduced such that fast Local

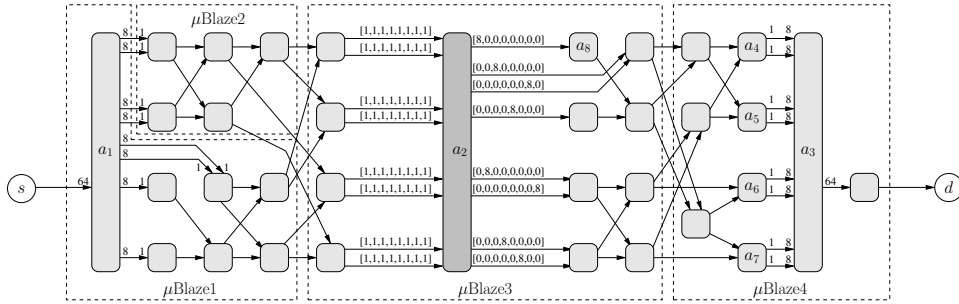


Fig. 16. Clustered multi-processor implementation of the two-dimensional *IDCT*. Inter-processor communication is realized via hardware *FIFO* links.

The latency improvement is slightly smaller with 34% (dynamic: 0.17 ms per 200 blocks, *QSS*: 0.11 ms per 200 blocks).

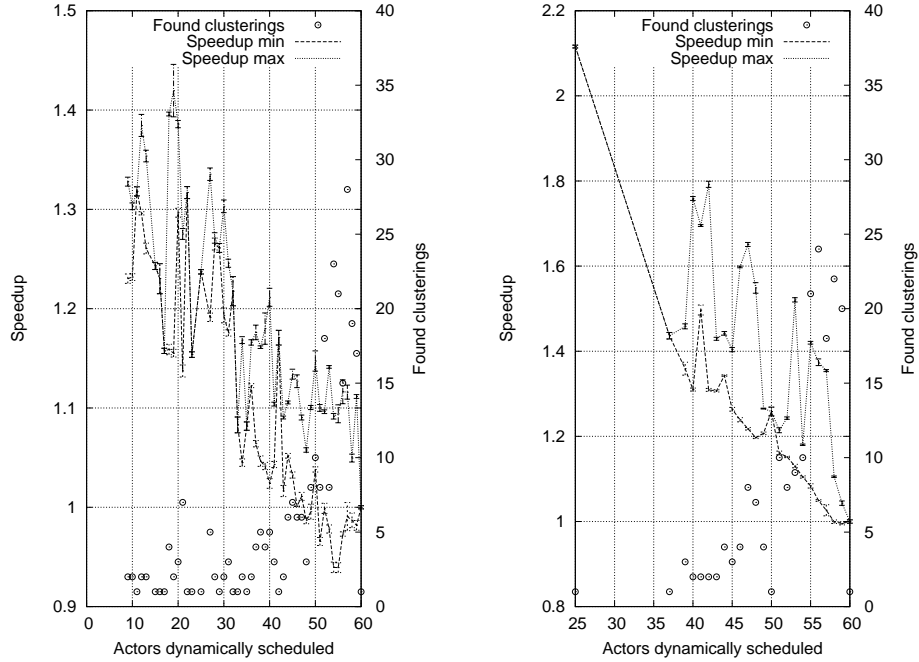
In order to more thoroughly evaluate the methodology, we also applied it to synthetic data flow graphs. We generated twenty synthetic *SDF* graphs with the *SDF3* tool [Stuijk et al. 2006b], with each graph containing sixty *SDF* actors and a configuration for the *SDF3* tool to minimize buffer capacities. This configuration of the *SDF3* tool generates buffer capacities which are problematic without the I/O buffer size adjustment algorithm. Indeed synthesizing the synthetic testcases without adjusting the I/O buffer sizes of the clusters resulted in 30% of the testcases deadlocking. The generated graphs are cyclic, i.e., they contain strongly connected components, with random but consistent *SDF* rates, initial tokens, and average in and out vertex degrees between two and nine.

After transforming such a generated graph into our model, we marked a fixed number of randomly chosen actors as dynamic by simply removing the classification tags added by the classification algorithm. It should be noted that due to the random marking of actors as dynamic, the more dynamic actors exist, the smaller the clusters become, i.e, in the worst case the remaining  $n$  static data flow actors are evenly scattered in the graph, eventually resulting in  $n$  clusters of size one.

The next step was to partition the static data flow actors into subgraphs satisfying the clustering condition from Definition 5.4. As an exhaustive evaluation is prohibitive, we transformed the clustering condition into a *SAT* problem and employed a *SAT* solver to detect valid clusters. In order to minimize the number of clusters in a clustering, the problem was parametrized by the maximum number of clusters allowed, e.g., one, to try to find a single cluster containing all static data flow actors. The *SAT* solver was started with this parameter varying from one to sixty to generate solutions with different numbers of dynamic actors remaining. This number is assigned to the x-axis in the Figures 17 - 18.

The circles in these figures denote how many clusterings (right y-axis) with a certain number of dynamic actors remaining where found. As can be seen the more remaining dynamic actor are permissible the more clusterings where found, e.g., in Figure 17(a) there are two clusterings with ten dynamically scheduled actors

Memory Buses (*LMB*) can be used.

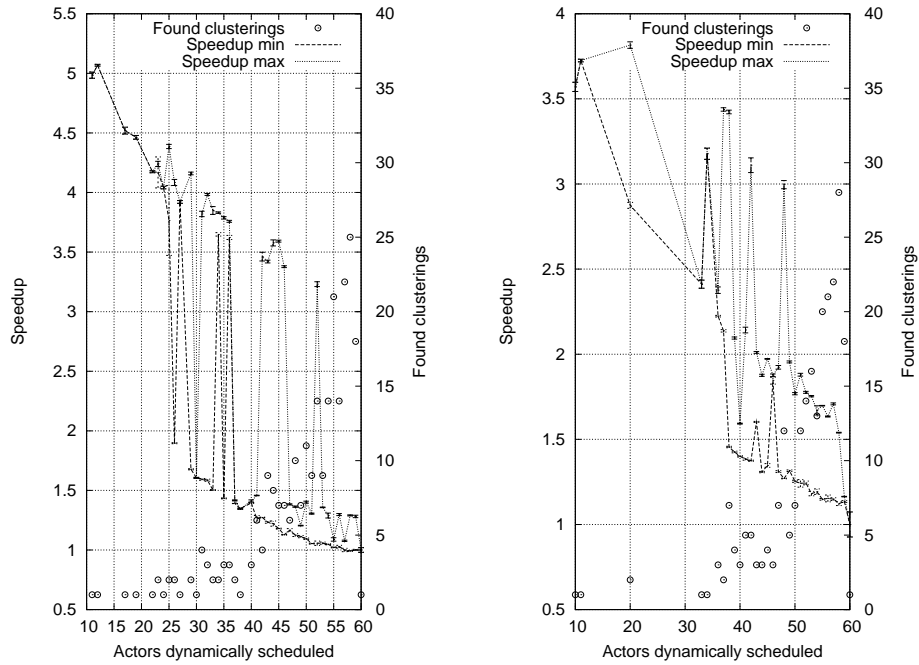


(a) Speedups obtained for clusterings of graph  $g_1$ . (b) Speedups obtained for clusterings of graph  $g_2$ .

Fig. 17. The diagram shows the speedups obtained by clusterings for two of the twenty synthetic data flow graphs.

remaining but 26 clusterings with 57 dynamically scheduled actors remaining. The number of remaining dynamic data flow actors theoretically ranges from two, the fully clustered case with a dynamic source actor and the dynamic actor corresponding to a single cluster containing all the static actors, to 60 actors, corresponding to a fully dynamic scheduled model. However, the SAT solver at most found clusterings with eight dynamic actors remaining.

Via this parametrization and the random marking of *SDF* actors as dynamic, we generated an average of two hundred unique clusterings for each synthetic data flow graph  $g_1 - g_{20}$  resulting in four thousand testcases. Each testcase corresponds to a clustering containing possibly multiple clusters. The speedup for each testcase was obtained by clustering, synthesizing (C++ code generation), compiling and executing the application on a Pentium 4 CPU, measuring the resulting runtime of the synthesized model (we took ten timing measurements) averaging the results and normalizing with the runtime of the model in the unclustered fully dynamically scheduled case. Depicted is the minimal (dashed line) and maximal (dotted dashed line) speedup (left y-axis) obtained for clusterings with a given number of remaining dynamic data flow actors (x-axis), e.g., in Figure 17(a) there are two clusterings with ten dynamically scheduled actors remaining where the first clustering achieved a speedup of 1.22 and the second a speedup of 1.32 where both measurements have an error of approximately 0.01. The observed speedup for highly static clusterings,



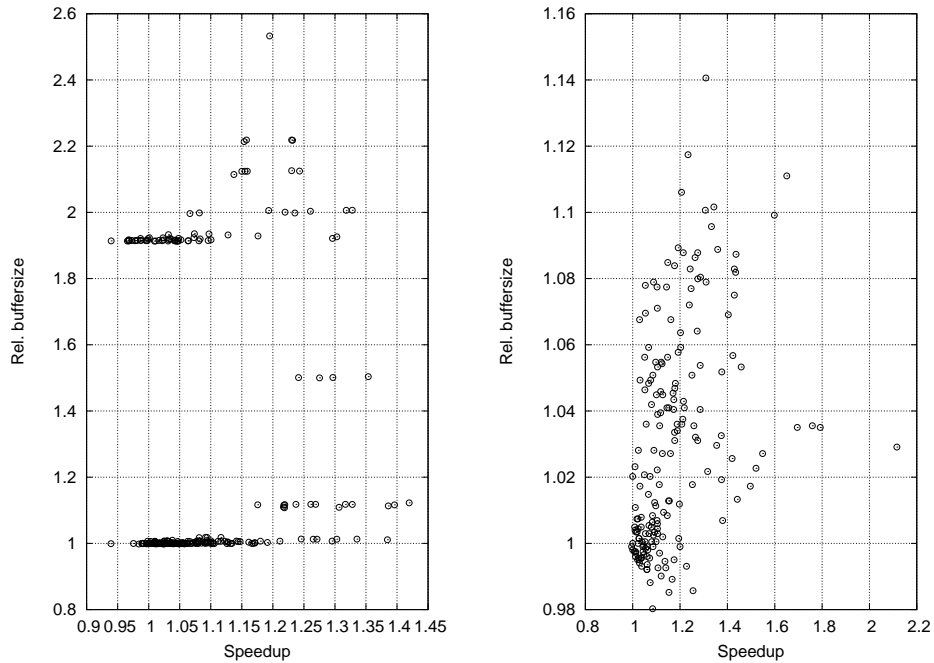
(a) Speedups obtained for clusterings of graph  $g_3$ . (b) Speedups obtained for clusterings of graph  $g_4$ .

Fig. 18. The diagram shows the speedups obtained by clusterings for two of the twenty synthetic data flow graphs.

i.e., clusterings with only ten dynamic actor remaining, was on average for the twenty synthetic graphs around 4 but for one graph two clusterings with speedup of factor 32 could be observed. This fact emphasizes the importance of selecting the right set of static actors to cluster. In the future we therefore want to use our design space exploration methodology [Keinert et al. 2009] to guide the selection of these actors. Finally in Figure 19 we have depicted for the synthetic graphs  $g_1$  and  $g_2$  for all their clusterings the speedup obtained versus the relative buffer space required after I/O buffer adjustment. Each circle in the figures is a clustering with its own speedup and relative buffer space increase. As can be seen again the selection of the set of static actors to cluster has a huge influence on the additional memory which is required. While there were always clusterings for all graphs  $g_1 - g_{20}$  even for high speedup clusterings were a buffer size increase of 20% percent was sufficient to prevent deadlock we have also seen bad clustering which required a fivefold increase of the memory requirements to prevent deadlock.

## 8. RELATED WORK

Models of Computation (*MoCs*) are an important concept in the design of embedded systems [Lee and Sangiovanni-Vincentelli 1998]. *MoCs* permit the use of efficient domain-specific optimization methods [Ziegenbein et al. 2002]. The advantages have been shown by many examples, e.g., for real time reactive systems



(a) Relative buffer increase vs speedup for clusterings of the graph  $g_1$ . (b) Relative buffer increase vs speedup for clusterings of the graph  $g_2$ .

Fig. 19. The diagram shows the relative buffer increase needed to prevent deadlocks for two of the twenty synthetic graphs.

[Baleani et al. 2005] and in the signal processing domain [Bhattacharyya et al. 2000; Bhattacharya and Bhattacharyya 2000]. An environment for modeling and simulating different and heterogeneous *MoCs* is provided by Ptolemy II [Lee 2004]. 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 *MoCs* 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 *MoCs* have been proposed in literature. FunState (Functions driven by state machines) [Strehl et al. 2001] 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 [Grötter et al. 2002; Baird 2005] is becoming the de facto standard for the design of digital hardware/software systems. SystemC permits the modeling of many different *MoCs*, but there is no unique representation of a particular *MoC* in SystemC. In order to identify different *MoCs* in SystemC, Patel et al. [Patel and Shukla 2005] have extended SystemC itself with different simulation kernels for *Communicating Sequential Processes* and *Finite State Ma-*

*chine MoCs* which also improves the simulation efficiency. Thus, the classification is again based on explicit statements. Other approaches supporting to model well-defined *MoCs* are also library-based and do not support actor classification, e.g., *YAPI* [de Kock et al. 2000], *DIF* [Hsu et al. 2005] and *SHIM* [Edwards and Tardieu 2005].

In contrast [Buck and Vaidyanathan 2000] can classify a subset of the `prim_model` actors, written in a C like notation, of Cocentric System Studio as *SDF/CSDF* actors by extracting CSDF phases from the source code. This extraction works analogous to our extraction of a *classification candidate*. However, it is unclear from [Buck and Vaidyanathan 2000] how branching control flow is handled. Our algorithm checks the classification candidate against all paths through the *CFG* of an actor.

The advantages of clustering *SDF* actors for the purpose of generating static schedules have been shown by Bhattacharyya et al. [Bhattacharyya and Lee 1993; Bhattacharyya et al. 1997; Pino et al. 1995]. However, these approaches only convert an *SDF* subgraph to an *SDF* actor and are intended to cluster the whole *SDF* graph into a single composite actor and thereby calculating a single processor schedule for the original *SDF* graph. Additionally, clustering is used for *MPSoC* scheduling by clustering actors which are later bound to a dedicated resource. In [Kianzad and Bhattacharyya 2006], a technique is developed to cluster data flow subgraphs to guide multiprocessor scheduling techniques towards solutions that provide lower schedule makespan (i.e., that minimize the time required to execute a single iteration of a data flow graph). However, this technique is limited to operating on homogeneous *SDF* graphs. Furthermore, the dynamic programming post optimization (*DPPO*) buffer minimization technique [Bhattacharyya and Murthy 1995] used in these scheduling approaches only operates on the buffers inside the cluster, as no I/O buffers exist after clustering the whole *SDF* graph into a single composite actor.

Scheduling *SDF* subgraphs within dynamic data flow graphs to decrease scheduling overhead is explored in [Buck 1993; Choi and Ha 1997]. However, in these approaches, the clustered *SDF* graphs are treated as atomic (pure *SDF*) actors, and therefore the clustering design space and the resulting schedules are more restricted compared to the approach presented in this paper. Finally, Vincentelli et al. [Sgroi et al. 1999] presented a quasi-static scheduling approach for equal conflict Petri nets. However, this technique is limited to pure equal conflict nets and exhibits exponential complexity in the number of conflicts. If only a subgraph exhibiting equal conflict semantics is scheduled with this technique, a best case environment is assumed, i.e., the feedback loop problem over the environment of the subgraph is neglected.

While there are many results [Stuijk et al. 2006a; Wiggers et al. 2007] concerning buffer estimation for *SDF* and *CSDF* systems, even in the presence of multiprocessor scheduling, buffer estimation for dynamic data flow graphs is *undecidable* in the general case [Buck 1993]. Scheduling *SDF* subgraphs within dynamic data flow graphs to decrease scheduling overhead is explored in [Buck 1993; Choi and Ha 1997]. However, in these approaches, the clustered *SDF* graphs are treated as atomic, i.e., they are embedded as pure *SDF* composite actors in the enclosing

dynamic data flow graph, and therefore no I/O buffer size adjustment must be performed as the composite actor is not created via some kind of clustering approach but is specified to have pure *SDF* actor semantics by the designer.

## 9. 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 presented 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 like clustering 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 data flow subgraphs in the application allows us to tackle the scheduling overhead problem encountered when modeling an application at fine granularity as necessary for design space exploration with heterogeneous mappings between actors and resources. The scheduling overhead is reduced by applying a generalized clustering approach that computes a quasi-static schedule for a given static data flow subgraph, thus reducing the scheduling overhead for one processor of an MPSoC while still accommodating a worst-case environment of the cluster. The clustering methodology presented in [Falk et al. 2008] has been extended to data flow graphs with bounded buffers, therefore enabling synthesis for embedded systems without dynamic memory allocation.

We have shown speedup benefits by a factor of 2 for the runtime of our Motion-*JPEG* decoder. For synthetic graphs we showed an average speedup by a factor of 4 for highly clustered graphs, e.g., graphs were no more than ten actors of sixty where dynamically scheduled. Future work will focus on optimized clustering techniques that build on our new framework for quasi-static scheduling. For example, we plan to explore algorithms for identification of actors that should be clustered in order to minimize the scheduling overhead, hence minimizing the number of states in the clustering FSM while maximizing the length of the static scheduling sequences.

## REFERENCES

- ABDI, S., PENG, J., YU, H., SHIN, D., GERSTLAUER, A., DOEMER, R., AND GAJSKI, D. 2003. *System-on-Chip Environment (SCE Version 2.2.0 beta): Tutorial*. UC Irvine, Irvine, CA. Tech. Rep. CECS-TR-03-41.
- BAIRD, M., Ed. 2005. *IEEE Standard 1666-2005 SystemC Language Reference Manual*. IEEE Standards Association, New Jersey, USA.
- BALEANI, M., FERRARI, A., MANGERUCA, L., SANGIOVANNI-VINCENTELLI, A., FREUND, U., SCHLENKER, E., AND WOLFF, H.-J. 2005. Correct-by-Construction Transformations across Design Environments for Model-Based Embedded Software Development. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)* (March 7–11). IEEE Computer Society.
- BHATTACHARYA, B. AND BHATTACHARYYA, S. 2000. Parameterized Dataflow Modeling of DSP Systems. In *Proc. of the International Conference on Acoustics, Speech, and Signal Processing*. Istanbul, Turkey, 1948–1951.

- BHATTACHARYYA, S. S., BUCK, J. T., HA, S., AND LEE, E. A. 1995. Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 42, 3 (Mar.), 138–150.
- BHATTACHARYYA, S. S. AND LEE, E. A. 1993. Scheduling synchronous dataflow graphs for efficient looping. *J. VLSI Signal Process. Syst.* 6, 3, 271–288.
- BHATTACHARYYA, S. S., LEUPERS, R., AND MARWEDEL, P. 2000. Software synthesis and code generation for signal processing systems. *IEEE Transactions on Circuits and Systems* 47, 9 (Sept.).
- BHATTACHARYYA, S. S., MURTHY, P., AND LEE, E. 1997. APGAN and RPMC: Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations. *Journal of Design Automation for Embedded Systems*.
- BHATTACHARYYA, S. S. AND MURTHY, P. K. 1995. Optimal parenthesization of lexical orderings for dsp block diagrams. In *In Proceedings of the International Workshop on VLSI Signal Processing*. IEEE press. 177–186.
- BILSEN, G., ENGELS, M., LAUWEREINS, R., AND PEPPERSTRAETE, J. 1996. Cyclo-Static Dataflow. *IEEE Transaction on Signal Processing* 44, 2 (Feb.), 397–408.
- BUCK, J. AND VAIDYANATHAN, R. 2000. Heterogeneous modeling and simulation of embedded systems in el greco. In *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*. ACM, New York, NY, USA, 142–146.
- BUCK, J. T. 1993. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.
- CHOI, C. AND HA, S. 1997. Software synthesis for dynamic data flow graph. In *Proceedings of the International Workshop on Rapid System Prototyping*.
- DE KOCK, E. A., ESSINK, G., SMITS, W. J. M., VAN DER WOLF, P., BRUNEL, J.-Y., KRUIJTZER, W. M., LIEVERSE, P., AND VISSERS, K. A. 2000. YAPI: Application Modeling for Signal Processing Systems. In *Proceedings of DAC*. 402–405.
- EDWARDS, S. A. AND TARDIEU, O. 2005. SHIM: A Deterministic Model for Heterogeneous Embedded Systems. In *Proceedings of EMSOFT*. 264–272.
- FALK, J., KEINERT, J., HAUBELT, C., TEICH, J., AND BHATTACHARYYA, S. 2008. A Generalized Static Data Flow Clustering Algorithm for MPSoC Scheduling of Multimedia Applications. In *EMSOFT'08: Proceedings of the 8th ACM international conference on Embedded software* (October 20–22).
- FZI Research Center for Information Technology 2007. *KaSCPar - Karlsruhe SystemC Parser Suite*. FZI Research Center for Information Technology. <http://www.fzi.de/sim/kascpa.html>.
- GEILEN, M. AND BASTEN, T. 2004. Reactive process networks. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*. 137–146.
- GRÖTKER, T., LIAO, S., MARTIN, G., AND SWAN, S. 2002. *System Design with SystemC*. Kluwer Academic Publishers.
- HSU, C. AND BHATTACHARYYA, S. S. 2007. Cycle-Breaking Techniques for Scheduling Synchronous Dataflow Graphs. Tech. Rep. UMIACS-TR-2007-12, Institute for Advanced Computer Studies, University of Maryland at College Park. Feb.
- HSU, C., KO, M., AND BHATTACHARYYA, S. S. 2005. Software synthesis from the dataflow interchange format. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*. Dallas, Texas, 37–49.
- JERRAYA, A. A., BOUCHHIM, A., AND PÉTROT, F. 2006. Programming Models and HW-SW Interfaces Abstraction for Multi-Processor SoC. In *Proceedings of DAC*. 280–285.
- KAHN, G. 1974. The semantics of simple language for parallel programming. In *IFIP Congress*. 471–475.
- KANGAS, T., KUKKALA, P., ORSILA, H., SALMINEN, E., HÄNNIKÄINEN, M., HÄMÄLÄINEN, T. D., RIIHIMÄKI, J., AND KUUSILINNA, K. 2006. UML-Based Multiprocessor SoC Design Framework. *ACM Transactions on Embedded Computing Systems* 5, 2 (May), 281–320.
- ACM Transactions on Computational Logic, Vol. V, No. N, May 2009.

- KEINERT, J., STREUBÜHR, M., SCHLICHTER, T., FALK, J., GLADIGAU, J., HAUBELT, C., TEICH, J., AND MEREDITH, M. 2009. SYSTEMCODESIGNER - An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications. *Transactions on Design Automation of Electronic Systems* 14, 1, 1–23.
- KIANZAD, V. AND BHATTACHARYYA, S. S. 2006. Efficient Techniques for Clustering and Scheduling onto Embedded Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* 17, 7, 667–680.
- LEE, E. A. 2004. Overview of the ptolemy project, technical memorandum no. ucb/erl m03/25. Tech. rep., Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, 94720, USA. July.
- LEE, E. A. AND MESSERSCHMITT, D. G. 1987. Synchronous Data Flow. *Proceedings of the IEEE* 75, 9 (Sept.), 1235–1245.
- LEE, E. A. AND SANGIOVANNI-VINCENTELLI, A. 1998. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17, 12 (Dec.), 1217–1229.
- PATEL, H. D. AND SHUKLA, S. K. 2005. 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*. Vol. 24. IEEE Press, Washington, D.C., 1261–1271.
- PINO, J. L., BHATTACHARYYA, S. S., AND LEE, E. A. 1995. A Hierarchical Multiprocessor Scheduling System for DSP Applications. In *Proc. Conf. on Signals, Systems, and Computers*. Vol. 1. 122–126.
- SGROI, M., LAVAGNO, L., WATANABE, Y., AND SANGIOVANNI-VINCENTELLI, A. 1999. Quasi-Static Scheduling of Embedded Software Using Equal Conflict Nets. In *Application and Theory of Petri Nets 1999. 20th International Conference, ICATPN'99*.
- STREHL, K., THIELE, L., GRIES, M., ZIEGENBEIN, D., ERNST, R., AND TEICH, J. 2001. FunState - An Internal Design Representation for Codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9, 4 (Aug.), 524–544.
- STUIJK, S., GEILEN, M., AND BASTEN, T. 2006a. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*. ACM, New York, NY, USA, 899–904.
- STUIJK, S., GEILEN, M., AND BASTEN, T. 2006b. SDF<sup>3</sup>: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*. IEEE Computer Society Press, Los Alamitos, CA, USA, 276–278.
- THOMPSON, M., NIKOLOV, H., STEFANOV, T., PIMENTEL, A., ERBAS, C., POLSTRA, S., AND DEPRETTERE, E. 2007. A Framework for Rapid System-level Exploration, Synthesis, and Programming of Multimedia MP-SoCs. In *Proceedings of CODES-ISSS'07*. 9–14.
- WIGGERS, M. H., BEKOOLJ, M. J. G., AND SMIT, G. J. M. 2007. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *DAC '07: Proceedings of the 44th annual conference on Design automation*. ACM, New York, NY, USA, 658–663.
- ZEBELEIN, C., FALK, J., HAUBELT, C., AND TEICH, J. 2008. Classification of General Data Flow Actors into Known Models of Computation. In *Proc. 6th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2008)*. Anaheim, CA, USA, 119–128.
- ZIEGENBEIN, D., RICHTER, K., ERNST, R., THIELE, L., AND TEICH, J. 2002. SPI- a system model for heterogeneously specified embedded systems. *IEEE Trans. on VLSI Systems*.