A Rule-Based Quasi-Static Scheduling Approach for Static Islands in Dynamic Dataflow Graphs

JOACHIM FALK, CHRISTIAN ZEBELEIN, CHRISTIAN HAUBELT, JÜRGEN TEICH,
University of Erlangen-Nuremberg

In this article, an efficient rule-based clustering algorithm for *static dataflow subgraphs* in a dynamic dataflow graph is presented. The clustered static dataflow actors are *quasi-statically scheduled*, in such a way that the global performance in terms of latency and throughput is improved compared to a dynamically scheduled execution, while avoiding the introduction of deadlocks as generated by naive static scheduling approaches. The presented clustering algorithm outperforms previously published approaches by a faster computation and more compact representation of the derived quasi-static schedule. This is achieved by a *rule-based* approach, which avoids an explicit enumeration of the state space. A formal proof of the correctness of the presented clustering approach is given. Experimental results show significant improvements in both, performance and code size, compared to a state-of-the-art clustering algorithm.

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

For the implementation of complex multimedia and signal processing applications, Multi-Processor System on Chip (MPSoC) architectures are becoming more and more important. However, due to the high degree of parallelism, programming these MPSoCs with traditional programming languages becomes quite error prone [Lee 2006]. Fortunately, dataflow graphs, which have been applied successfully to application modeling in these domains, expose inherent application parallelism and thus map well into the MPSoC world. Thereby, the overall performance of dataflow applications mapped onto MPSoCs is often highly affected by the binding of individual dataflow actors onto the different processing cores.

Once the mapping of the actors onto the MPSoC has been determined, there are still problems to efficiently schedule the actors bound onto the same processing core. A straight forward scheduling idea that postpones all scheduling decisions to runtime, i.e., *dynamic scheduling*, can be detrimental to system performance. In particular, if the scheduled actors are very fine grained, the scheduling overhead incurred by runtime decisions takes up a noticeable amount of the total computational power. This is especially true if the multimedia or signal processing applications contain parts that can be scheduled statically, which is often the case.

As *static schedules* do not contain any runtime decisions, statically scheduling these parts seems to be another option. However, as the resulting subsystem has to communicate with dynamic parts of the application and must cope with dynamic requests from remote processing cores, computing static schedules is not viable in general.

The scheduling overhead problem could be mended by choosing an appropriate level of granularity, i.e., merging as much functionality into a single actor such that the computation costs dominate the scheduling overhead. Ideally, in such a merging step, a *quasi-static schedule* is computed and implemented inside the newly created composite actor. In a quasi-static schedule, runtime decisions are followed by statically scheduled sequences. Hence, the number of runtime decisions is less than in dynamic schedules and, as a consequence, the scheduling overhead is reduced. The runtime decisions are then used to respond correctly to dynamic requests from the environment of the composite actor.

Such a merging step is equivalent to *manually clustering* the dataflow graph, i.e., static dataflow actors are clustered into a single composite actor and the static dataflow actors are scheduled for a single processing core. Consequently, all these actors have to be bound onto the same processing core afterwards. However, this constraints the design space by removing mapping options. If the mapping itself is part of the synthesis step – as is the case in nearly all system synthesis tools [Gerstlauer et al. 2009] – an appropriate level of granularity can no longer be determined in advance. Thus, clustering has to be performed automatically after binding actors of the dataflow graph onto the processing cores of the MPSoC.

In this article, an *automatic clustering* algorithm for static dataflow actors mapped onto the same processing core is proposed. As these static dataflow actors might communicate with dynamic actors, a pure static schedule cannot be determined in general. The presented quasi-static scheduling approach assumes a worst case behavior of the environment of the static dataflow graph. This guarantees that no deadlocks are introduced into the implementation while at the same time the overall performance in terms of latency and throughput is improved.

In contrast to previous embedded software synthesis approaches based on clustering, the enumeration of the state space is avoided. For this purpose, state sets are represented *implicitly* and state transitions are defined by *rules*. Hence, the proposed clustering approach is able to schedule larger systems and generates more compact schedules. The latter results in reduced code sizes of the generated software. However, these advantages are paid by the price of a slightly increased computation time for runtime decisions.

The remainder of this article is organized as follows: After formally defining the problem and giving a motivating example in Section 2, related work will be reviewed in Section 3. Section 4 presents the proposed clustering algorithm, whereas its correctness is proven in Section 5. Experimental results presented in Section 6 will show the performance improvements by applying the proposed clustering algorithm to heterogeneous dataflow graphs. Moreover, a comparison to a state-of-the-art clustering approach will show the efficiency of the proposed clustering algorithm in terms of scalability and code size reduction.

2. PROBLEM DEFINITION

In this section, the problem of clustering static dataflow subgraphs embedded in dynamic dataflow graphs is formally defined. The overall goal is the reduction of scheduling overhead due to non-essential checks at runtime. Before considering the MPSoC scheduling problem, the single processor scheduling problem is discussed first, i.e., a single multimedia application is bound onto a single processing core. The application model is given as a general *dataflow graph*:

Definition 2.1 (*Dataflow Graph*). A *dataflow graph* is a directed graph $g = (A, C)$, where the set of vertices A represents the *actors* and the set of edges $C \subseteq A \times A$ represents the *channels*. Additionally, a *delay function* $d : C \to \mathbb{N}_0$ is given. It assigns to each channel $(a_{\mathrm{src}}, a_{\mathrm{dst}}) \in C$ a non-negative number of initial tokens.

Actors in a dataflow graph perform the actual computation by so called *firings*. An actor can *fire* if a sufficient number of tokens is available on its input channels (incoming edges). When an actor $a \in A$ fires, it consumes tokens from its input channels and produces tokens onto its output channels (outgoing edges). The behavior of an actor might be either *static* or *dynamic*. Static actors consume and produce tokens with constant (or periodically constant) rates. In contrast to this, dynamic actors have variable consumption and production rates. The precondition, which decides if an actor can fire or not, is called the *firing rule*. For static actors, the firing rule consists only of the required number of tokens per incoming channel. Although the proposed clustering algorithm can handle static actors with periodically constant rates (cyclo-static dataflow, [Bilsen et al. 1996]), in the following only static actors

with constant rates (synchronous dataflow, [Lee and Messerschmitt 1987b]) are considered for the sake of clarity.

When generating single processor embedded software from dataflow graphs, one of the most important tasks is computing a schedule. One option is to generate dynamic schedules. In this case, the firing rules of the actors are checked at runtime following the chosen scheduling policy. As soon as a firing rule is satisfied, the corresponding actor is fired. If, however, the dataflow graph is a static dataflow graph, consumption and production rates are known at compile time. In this case, generating a static schedule is another option. A static schedule is a sequence of actor firings, which is periodically executed.

For static scheduling computation in static dataflow graphs, the number of firings of each actor has to be known. Typically these numbers are represented by the so called *repetition vector* $\boldsymbol{\mu} = (\mu_{a_1}, \mu_{a_2}, \ldots \mu_{a_m})$, which can be calculated by the *balance equations* that can be derived from the graph structure. An SDF graph is called *consistent* if the balance equations have a non-trivial integer solution, i.e., a solution where all elements in the repetition vector are positive. In this case, a static schedule τ may exist such that the graph has the same state before and after executing τ.[1] Note that the state of a static dataflow graph is given by the number of tokens stored on each channel. Given a consistent schedule τ, each actor $a_i \in A$ is fired exactly μ_{a_i} times. Note that performing dynamic scheduling for static dataflow graphs results in unwanted scheduling overhead due to the non-essential checking of firing rules.

Nowadays, typical multimedia applications are neither pure static dataflow nor fully dynamic dataflow. In such scenarios, static dataflow subgraphs are embedded in more general (dynamic) dataflow graphs. Dynamically scheduling such graphs might again result in unnecessary scheduling overhead due to non-essential checking of static dataflow actors. On the other hand, statically scheduling the static dataflow actors is too optimistic as these static actors are connected to a dynamic environment. Indeed, the short example below will illustrate that static scheduling might introduce deadlocks in the application. Thus, the question is how to exploit the knowledge about static subgraphs during schedule generation without introducing deadlocks.

In the following, an approach based on clustering static dataflow actors into a single *composite actor* a_c will be proposed. This composite actor a_c schedules the firings for the clustered static dataflow actors in such a way that non-essential firing rule checks are reduced, while at the same time avoiding the introduction of deadlocks. In the general case, the composite actor is a dynamic dataflow actor whose firing rules depend on the availability of tokens on its input channels and some internal state. The internal state captures firings of clustered static dataflow actors already performed.

Let A_S be the subset of of all static actors from the set of actors A of a dynamic dataflow graph. Then, the induced static subgraph is the graph $g_S = (A_S, C_S)$ that contains all static actors $a \in A_S$ and only those channels $c \in C$, which connect static actors, i.e., $C_S = \{c = (a_0, a_1) \in C \mid a_0, a_1 \in A_S\}$. This subgraph may not be connected and even if it is connected the presented *quasi-static scheduling algorithm* may not be applicable to it. The presented algorithm requires the *clustering condition* (cf. Definition 4.10 on page 17) to be satisfied before it can be applied to a subgraph. However, it is always possible to decompose g_S into subgraphs which satisfy the *clustering condition* and to which the presented algorithm can be applied separately. In the following we will assume that g_S already satisfies the clustering condition.

Example 2.2. Given the dataflow graph in Fig. 1. Actor a_4 is a dynamic actor whereas a_1, a_2 and a_3 are static actors. Hence, the induced static subgraph is $g_S = (A_S, C_S)$ with $A_S = \{a_1, a_2, a_3\}$ and $C_S = \{(a_1, a_2), (a_2, a_1), (a_2, a_3), (a_3, a_2)\}$.

[1] Even if the SDF graph is *consistent*, insufficient initial tokens on the edges forming cycles in the graph may prevent the existence of a valid schedule τ due to deadlock.

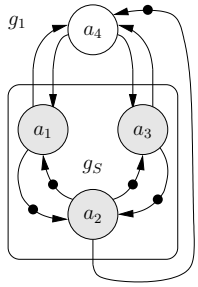


Fig. 1. Heterogeneous dataflow graph with a single dynamic dataflow actor a_4 and three static dataflow actors a_1, a_2, a_3. The constant consumption and production rates of the static dataflow actors are annotated to the edges. Edges without annotation have a consumption or production rate of one. The delays d are depicted as black tokens on the edges. The static dataflow actors can be clustered into a single composite actor a_c reducing scheduling overhead.

In the following, it is assumed that the induced static dataflow graph g_S is a connected graph. In this case the static dataflow graph g_S can be replaced by a composite actor a_c. The result is a new dataflow graph $\tilde{g} = (\tilde{A}, \tilde{C})$ containing only the composite actor and the remaining dynamic actors, i.e., $\tilde{A} = A - A_\mathrm{S} + \{a_\mathrm{c}\}$. The set of channels $\tilde{C} = C - C_\mathrm{c} + C_\mathrm{n}$ consists of channels only connecting dynamic actors, i.e., $C - C_\mathrm{c}$ with $C_\mathrm{c} = \{(a_i, a_j) \in C \mid a_i \in A_\mathrm{S} \vee a_j \in A_\mathrm{S}\}$ being the set of channels connected to at least one actor in the subgraph, and new channels C_n connected to the composite actor a_c, i.e., $C_\mathrm{n} = \{(a_i, a_j) \in \tilde{A} \times \tilde{A} \mid (a_i = a_\mathrm{c} \implies \exists (a', a_j) \in C : a' \in A_\mathrm{S}) \vee (a_j = a_\mathrm{c} \implies \exists (a_i, a') \in C : a' \in A_\mathrm{S})\}$. The delay function d is adapted accordingly.

In the general case all actors in $\tilde{g}$ are dynamic. Hence, they must be scheduled dynamically at runtime. Therefore, the scheduling overhead can only be reduced by the schedule implemented in the composite actor. Note that the scheduling overhead decreases with the length of static firing sequences implemented in the composite actors. This is due to the fact that within a static sequence, the checking of firing rules, which is the main reason for scheduling overhead, can be omitted. Obviously, the biggest reduction is encountered by implementing a pure static schedule in the composite actor. However, in the general case this will introduce deadlocks as will be shown in the following example.

Example 2.3. For the dataflow graph shown in Fig. 1, the resulting dataflow graph $\tilde{g}$ after clustering consists of two actors, namely a_4 and a_c, and five edges. The repetition vector of the induced static dataflow subgraph is $\boldsymbol{\mu}_\mathrm{c} = (1, 1, 1)$. A possible fully static single processor schedule implemented by the composite actor a_c is $\langle a_1 a_2 a_3 \rangle$. To execute this static schedule atomically (otherwise it would not be a static schedule), at least one token must be available on each input channel (a_4, a_1) and (a_4, a_3). Assuming that firing a_4 will not change the number of tokens in the graph, i.e, a_4 only redistributes tokens from channels (a_1, a_4), (a_3, a_4) and (a_2, a_4) to channels (a_4, a_1) and (a_4, a_3), the static schedule can never fire as it requires a total of two tokens on the composite actor's inputs instead of the single token present outside of g_S. Thus, implementing a static schedule might introduce deadlocks, even if the original system has been deadlock-free, which can be easily checked using self-scheduled execution semantics, where each actor fires as soon as a sufficient number of tokens is available on its input channels.

The above example has shown that implementing a static schedule in the composite actor might lead to invalid implementations due to the unknown behavior of the environment of the induced static dataflow subgraph. As a consequence, when solving the problem of clustering a static dataflow graph embedded in a dynamic dataflow graph, a worst case behavior has to be assumed for its environment. The worst case, however, occurs if each

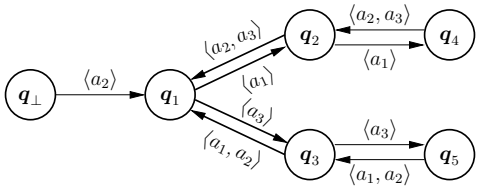


Fig. 2. Quasi-static schedule for the induced static subgraph g_S from Fig. 1 represented as an FSM with six states $\boldsymbol{q}_\perp, \boldsymbol{q}_1, \boldsymbol{q}_2, \ldots \boldsymbol{q}_5$ and nine transitions. Static schedule sequences are annotated to the transitions. Input token requirements can be derived from these sequences and are omitted here.

token produced by the composite actor a_c is again required as input to a_c, i.e., so called *tight feedback loops* are assumed.

Example 2.4. A schedule that does not introduce deadlocks when clustering the static dataflow actors $A_\mathrm{S} = \{a_1, a_2, a_3\}$ from Fig. 1 is shown in Fig. 2. The schedule is represented as a finite state machine (FSM). In the initial state $q_\perp$, actor a_2 can be fired without firing any other actor of the cluster, due to the initial tokens contained in the cluster. In state q_1, depending on the availability of tokens on the cluster input channels, either a_1 or a_3 can be fired, leading to state q_2 or q_3, respectively. In state q_2, the situation is slightly different: Again, either a_1 or a_3 can be fired depending on the availability of tokens. However, if a_3 is executed, a_2 is also fired. Note that this statically scheduled firing sequence returns the FSM to state q_1. Obviously, the FSM does not introduce deadlocks although it contains statically scheduled sequences, as it is always possible to either execute a_1 or a_3 depending on the availabilty of tokens on the cluster input channels.

Note that each path in the FSM starting from $\boldsymbol{q}_\perp$ corresponds to a valid schedule sequence. Which path is selected depends upon runtime decisions based on the availability of tokens. Furthermore, for any cyclic path in the FSM and any actor a the accumulated number of actor firings for actor a in the cycle correspond to the firings specified by the repetition vector $\boldsymbol{\mu}_\mathrm{c}$ for actor a.

The resulting schedule is a so called *quasi-static schedule*. In quasi-static schedules, runtime decisions are followed by static sequences of actor firings. The runtime decisions serve the purpose to react to dynamic effects of the environment of the induced static dataflow graph, thus avoiding deadlocks. The static sequences are used to reduce the number of firing rule checks at runtime, thus reducing scheduling overhead. In summary, *clustering* can be defined as follows:

Definition 2.5 (*Clustering*). Given a dataflow graph g and the set of static actors A_S. Clustering replaces the induced static dataflow graph g_S by a single actor a_c, called *composite actor*, which *quasi-statically schedules* the firings of the static dataflow actors A_S.

Before describing in detail the computation of the cluster schedules in the following sections, it should be noted that clustering can be also used in the context of MPSoCs.[2] In this case, the binding $\beta : A \to R$ that assigns each actor $a \in A$ to a processing core $r \in R$ of the MPSoC must be taken into account. As a consequence, for each processing core $r \in R$ a static dataflow subgraph $g_{\mathrm{S},r}$ induced by the static actors $a \in A_{\mathrm{S},r} = \{a \in A_\mathrm{S} \mid \beta(a) = r\}$ bound to r is computed and used in the clustering. As the proposed clustering algorithm assumes a worst case environment, the generated schedules allow to respond to dynamic requests from remote processing cores. In this sense, single processor scheduling is just a special case where all actors are bound to a single resource.

[2]Note that the performance gain does not only depend on the scheduling, but also on the communication overhead in MPSoCs.

3. RELATED WORK

When mapping dataflow applications onto MPSoC platforms, the overall performance of the implementations is highly dependent on the actual mapping of actors to processing cores together with the implied communication load. In the past, different approaches of optimally mapping a dataflow application onto a given MPSoC platform have been proposed, e.g., Daedalus [Thompson et al. 2007], SystemCoDesigner [Haubelt et al. 2007], PeaCE/HOPES [Ha et al. 2007; Kwon et al. 2008], and DOL [Thiele et al. 2007]. A good overview on model-based software design flows for MPSoCs is given in [Haid et al. 2009].

Beside the mapping, the scheduling on each individual processing core can also have a huge impact on the resulting performance. In the domain of single processor scheduling of dataflow applications many results have been published. For instance, efficient single processor scheduling algorithms [Bhattacharyya et al. 1995; Hsu and Bhattacharyya 2007] exist for *synchronous dataflow (SDF)* graphs [Lee and Messerschmitt 1987b], a widely accepted static dataflow model. *Cyclo-static dataflow (CSDF)* graphs [Bilsen et al. 1996] could be scheduled similarly by first unfolding the CSDF graph according to its hyperperiod. This, however, may exponentially increase the number of actors. Unfortunately, these static modeling approaches are insufficient to model real world multimedia and signal processing applications. These applications require a model, which supports heterogeneous dataflow graphs that also contain actors with dynamic behavior like *Kahn processes*. Furthermore, simply applying the static scheduling methodologies developed for self-contained SDF and CSDF graphs to SDF and CSDF subgraphs embedded in general dataflow graphs can introduce deadlocks into the resulting system in the general case.

The advantages of clustering SDF subgraphs for the purpose of generating static schedules have been shown in [Bhattacharyya and Lee 1993], which introduced the *Pairwise Grouping of Adjacent Nodes (PGAN)* clustering algorithm for constructing lexical orderings for later conversion into *single appearance schedules*. However, to detect the feasibility of a clustering operation, this algorithm uses the corresponding *Acyclic Precedence Graph (APG)* [Lee and Messerschmitt 1987a] of the SDF subgraph, a representation that can grow exponentially in the number of actors. Due to the restriction that the APG can only be derived for SDF graphs, more general dataflow graphs cannot be clustered using this algorithm. An improvement of PGAN has been presented by Bhattacharyya et al. [Bhattacharyya et al. 1997] with the *Acyclic Pairwise Grouping of Adjacent Nodes (APGAN)* algorithm. This algorithm no longer requires the construction of the APG thus avoiding the exponential blowup problem. However, the algorithm is only applicable to acyclic graphs. Another complementary heuristic for clustering is the top-down algorithm *Recursive Partitioning by Minimum Cuts (RPMC)* also presented in this work. Both APGAN and RPMC could in principle be used for clustering heterogeneous dataflow systems due to the restriction that only acyclic graphs are handled. This evades the problem of feedback loops as considered in the presented approach in the article at hand. 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 dataflow subgraphs to guide multiprocessor scheduling techniques towards low latency schedules. However, this technique is limited to operating on homogeneous SDF graphs (SDF graphs with unit consumption and production rates). In [Pino et al. 1995] a heuristic for SDF graph clustering is presented which avoids the exponential growth problem of PGAN but is also limited to generating SDF actors.

Scheduling SDF subgraphs within dynamic dataflow graphs to decrease scheduling overhead is explored in [Buck 1993; Choi and Ha 1997]. However, in these approaches, the clustered SDF subgraphs are treated as atomic (pure SDF) actors, and therefore the clustering design space and the resulting schedules are more restricted compared to those associated with the approach that is presented in this article. 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.

Process merging while keeping throughput constraints has been explored by Stefanov et al. [Meijer et al. 2010]. However, the presented methodology is only applicable to acyclic graphs, thus, the deadlock problem handled in this article does not appear. Furthermore, these acyclic graphs are not arbitrary acyclic graphs but graphs induced by nested loop programs.

Plishker et al. [Plishker et al. 2009a] have presented a scheduling methodology for dynamic dataflow graphs which improves on the simple round-robin scheduler. The actors in this dynamic graphs are constrained to switch between static modes. If an actor remains in a fixed mode it behaves like an SDF actor. The approach has been extended in [Plishker et al. 2009b] to also exploit static sequences of modes, e.g., also exploiting a kind of CSDF behavior of the dynamic actors. For certain mode assignments consistent SDF subgraphs form in the dynamic dataflow graph. If the SDF subgraph exhibits multirate behavior an optimized schedule compared to a round-robin schedule is constructed. In this optimized schedule actors are checked for activatability in correct proportion to each other. Plishker's approach is more general, as it can handle certain kinds of dynamic actors, but also more limited as it still needs to check the activatability of each actor firing. In contrast, the approach presented here can detect whole chains of actor firings which can be executed statically.

Furthermore, Janneck et al. [Gu et al. 2011] have presented a methodology for detecting *statically schedulable regions (SSRs)* in general CAL [Eker and Janneck 2003] programs. This analysis extends the idea of finding islands of static SDF or CSDF actors in a general dataflow graph to also consider only parts of actors as static if their static behavior is only exhibited for a subset of their ports. This paper describes the detection algorithm for SSRs but doesn't provide the details about the actual exploitation of these regions for synthesis.

Later on, in [Falk et al. 2008] many of the above mentioned limitation have been overcome by extending the static scheduling analysis [Lee and Messerschmitt 1987b; Bhattacharyya et al. 1995; Hsu and Bhattacharyya 2007] for self-contained SDF and CSDF graphs to SDF and CSDF subgraphs. The analysis in [Falk et al. 2008] produces a *quasi-static schedule* encoded as a so-called *cluster finite state machine (cluster FSM)*. This approach is the most similar approach to the one presented in the article at hand. However, representing schedules by FSMs has some severe disadvantages. Looking again at the schedule shown in Fig. 2, it can be observed that the scheduling sequences $\langle a_1 \rangle$, $\langle a_1, a_2 \rangle$, $\langle a_3 \rangle$ and $\langle a_2, a_3 \rangle$ occur multiple times in the FSM. This is due to the representation of the partial order of the scheduling sequences by means of the explicit enumeration of all possible sequences in the FSM. This leads to the well known exponential state–space explosion problem, e.g., adding another actor a_5 in the manner of a_1 and a_3 adds only two independent scheduling sequences $\langle a_5 \rangle$ and $\langle a_2, a_5 \rangle$. However, the resulting FSM consists of 20 states and approx. 40 transitions.

Solutions for partial order reduction are known [Alur et al. 2001] from the verification domain. However, this approach cannot be applied to the problem under consideration as only one possible sequence from the partial order is considered, relying on the equivalence of all such sequences for this simplification. This reduction cannot be performed for clustering as all such sequences must be executable to satisfy a worst case environment of the cluster.

Finally, the presented article is an extension of [Falk et al. 2011] which solves the schedule representation problem of [Falk et al. 2008]. The presented approach in this article has two major contributions: (1) the *construction* of the quasi-static schedules for clustered static dataflow subgraphs is *faster* than the approach proposed in [Falk et al. 2008] due to an implicit representation of the states in the schedule. (2) the generated schedules lead to

more *compact code* compared to the approach in [Falk et al. 2008] during embedded software synthesis due to a rule-based representation of the quasi-static schedule, thus alleviating the state space explosion problem. In summary, the approach presented in the article at hand is able to handle more complex systems compared to previous approaches. One drawback of the presented approach compared with the cluster FSM approach presented in [Falk et al. 2008] is, however, that the software runtime might slightly increase due to the more complex checking of firing rules. In contrast to [Falk et al. 2011] a more complex example to detail the finer points of the clustering algorithm is used in Section 4. A correctness proof for the presented algorithm is given in Section 5. In Section 6, complex case studies and benchmarks are presented to evaluate the proposed algorithm.

4. CLUSTERING OF STATIC DATAFLOW SUBGRAPHS

In this section, a clustering approach is presented, which replaces a given static dataflow subgraph g_S by a composite actor a_c based on the computation of a quasi-static schedule (QSS) for a_c such that a_c does not introduce deadlocks into the system.[3] In order to guarantee this property, tokens produced by the composite actor a_c are assumed to be required again as input to a_c.

To identify the tokens consumed and produced by the composite actor a_c, the static dataflow actors $a \in A_\mathrm{S}$ of the static dataflow subgraph g_S are tagged as *input actors* $A_\mathrm{I} \subseteq A_\mathrm{S}$ and *output actors* $A_\mathrm{O} \subseteq A_\mathrm{S}$. Note that A_I and A_O may not be disjoint sets. Input actors are static actors having at least one incoming channel from a non-static actor, i.e., $A_\mathrm{I} = \{a \in A_\mathrm{S} \mid \exists(\tilde{a}, a) \in C \wedge \tilde{a} \notin A_\mathrm{S}\}$. Analogously, output actors are static actors having at least one outgoing channel to a non-static actor, i.e., $A_\mathrm{O} = \{a \in A_\mathrm{S} \mid \exists(a, \tilde{a}) \in C \wedge \tilde{a} \notin A_\mathrm{S}\}$.

For the sake of simplicity, it is assumed in the following that each input actor is connected to exactly one incoming channel from a non-static actor, and each output actor is connected to exactly one outgoing channel to a non-static actor. Note that this assumption can be made without loss of generality, as this property can be enforced by inserting additional static actors having no net effect on the consumption and production rates.

The proposed clustering approach basically works in two steps: In the first step, the state space given by static dataflow actor firings is implicitly constructed using a *rule-based approach*. The result of the first step are rules that specify possible actor firings in the static dataflow subgraph in dependence of previous actor firings. Note that the possible actor firings are still unscheduled (but represent static sequences). Hence, in a second step, a feasible single processor schedule is computed resulting in static sequences without dynamic scheduling decisions [Falk et al. 2011].

4.1. Implicit State Space Representation

As token productions must not be postponed, the proposed algorithm operates on the so called *input/output dependency function*, which encodes the minimum number of input actor firings required for a given number of output actor firings. With this information it is known how many tokens are required on the input channels in order to produce output tokens. Consequently, an actor can be fired as soon as the specified amount of tokens is available. More formally, the input/output dependency function is defined as follows:

Definition 4.1 (*Input/Output Dependency Function*). Given a static dataflow subgraph g_S, its *input/output dependency function* $\mathbf{dep}_\mathrm{io} : A_\mathrm{O} \times \mathbb{N}_0 \to {\mathbb{N}_0}^{|A_\mathrm{I}|}$ is a vector-valued function that associates with each output actor $a_o \in A_\mathrm{O}$ and a given number of firings $n \in \mathbb{N}_0$ the minimum number of input actor firings $(q_{a_{i,1}}, q_{a_{i,2}}, \ldots, q_{a_{i,|A_\mathrm{I}|}})$.

[3]The original system is assumed to be deadlock-free. This, however, cannot be detected due to the undecidability of deadlock freedom for general dataflow graphs.

Table I. Input/output dependency function values $(q_{a_1}, q_{a_3}) = \mathbf{dep}_{\mathrm{io}}(a, n)$ for subgraph g_S from Fig. 1 and corresponding input/output dependency states $(q_{a_1}, q_{a_2}, q_{a_3}) = \boldsymbol{q}_{\mathrm{io}}$.

	$\mathbf{dep}_{\mathrm{io}}(a, n)$			Corresponding $\boldsymbol{q}_{\mathrm{io}}$ for $\mathbf{dep}_{\mathrm{io}}(a, n)$		
n	$a = a_1$	$a = a_2$	$a = a_3$	$a = a_1$	$a = a_2$	$a = a_3$
0	$(0, 0)$	$(0, 0)$	$(0, 0)$	$(0, 1, 0)$	$(0, 0, 0)$	$(0, 1, 0)$
1	$(1, 0)$	$(0, 0)$	$(0, 1)$	$(1, 1, 0)$	$(0, 1, 0)$	$(0, 1, 1)$
2	$(2, 0)$	$(1, 1)$	$(0, 2)$	$(2, 1, 0)$	$(1, 2, 1)$	$(0, 1, 2)$
3	$(3, 1)$	$(2, 2)$	$(1, 3)$	$(3, 2, 1)$	$(2, 3, 2)$	$(1, 2, 3)$
			...			

Hence, to perform a requested number of n firings of a given output actor $a_o \in A_{\mathrm{O}}$, at least $\mathbf{dep}_{\mathrm{io}}(a_o, n)$ input actor firings are required.

Example 4.2. Considering again the dataflow graph given in Fig. 1, the static dataflow subgraph $A_S = \{a_1, a_2, a_3\}$ has two input and three output actors, i.e., $A_{\mathrm{I}} = \{a_1, a_3\}$ and $A_{\mathrm{O}} = \{a_1, a_2, a_3\}$. In order to fire actor a_2 once, $\mathbf{dep}_{\mathrm{io}}(a_2, 1) = (0, 0)$ input actor firings are required, i.e., the first firing of actor a_2 can be performed without firing any input actors, due to the initial tokens contained in the cluster. However, in order to fire a_2 for a second time, $\mathbf{dep}_{\mathrm{io}}(a_2, 2) = (1, 1)$ input actor firings are required, i.e., both actor a_1 and a_3 have to be fired once. Further values are depicted in Table I.

Given $\mathbf{dep}_{\mathrm{io}}$, the so-called *input/output dependency states* Q_{io} can be calculated. Each input/output state $\boldsymbol{q} = (q_{a_{i,1}}, q_{a_{i,2}}, \ldots, q_{a_{i,|A_I|}}, q_{a_{o,1}}, q_{a_{o,2}}, \ldots, q_{a_{o,|A_{\mathrm{O}}-A_{\mathrm{I}}|}}) \in Q_{\mathrm{io}}$ is a possible state of execution of the static actors A_{S} in a self scheduled execution and is additionally constraint to provide the maximum possible number of output actor firings for a minimum number of required input actor firings. This constraint stems from the fact that the clustering algorithm has to assume a worst case behavior of the subgraph's environment, and thus, must neither consume more input tokens than necessary nor postpone the production of output tokens.

There may be one exception, namely the initial state $\boldsymbol{q}_\perp = \mathbf{0} \in Q_{\mathrm{io}}$ (the all zero vector) which is explicitly added to the set of input/output dependency states. This is due to the fact that sufficient initial tokens stored on the internal channels may permit the firing of cluster output actors without firing any cluster input actors (cf. Fig 2). However, this situation can occur at most once per cluster, as subsequent static sequences always produce the maximum number of output tokens from a minimum number of input tokens, and thus, tokens will not accumulate on internal channels.

Due to Definition 4.1, the number of input actor firings from the input/output dependency function $\mathbf{dep}_{\mathrm{io}}$ is already minimal. Therefore, in order to derive an input/output dependency state from $\mathbf{dep}_{\mathrm{io}}$, only the output actor firings have to be maximized. More formally, the initial input/output dependency states are defined as follows:[4]

Definition 4.3 (*Input/Output Dependency States*).

$$\begin{aligned} Q_{\mathrm{io}} \;=\; \{\boldsymbol{q}_\perp\} \cup \{(\mathbf{dep}_{\mathrm{io}}(a_o, n), q_{a'_{o,1}}, q_{a'_{o,2}}, \ldots, q_{a'_{o,|A_{\mathrm{O}}-A_{\mathrm{I}}|}}) \\ \mid a_o \in A_{\mathrm{O}}, n \in \mathbb{N}_0, \\ \forall a'_o \in A_{\mathrm{O}} - A_{\mathrm{I}} : q_{a'_o} = \max\{n' \in \mathbb{N}_0 \mid \\ \mathbf{dep}_{\mathrm{io}}(a'_o, n') \leq \mathbf{dep}_{\mathrm{io}}(a_o, n)\}\}. \end{aligned}$$

Note that the maximum operation calculates for each output actor a'_o the maximum number of firings which can be performed by at most $\mathbf{dep}_{\mathrm{io}}(a_o, n)$ firings of input actors.

[4]In the following, comparisons between two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ will be defined as follows: $\boldsymbol{a} \leq \boldsymbol{b} \iff \forall i : a_i \leq b_i$ and $\boldsymbol{a} < \boldsymbol{b} \iff \boldsymbol{a} \leq \boldsymbol{b} \wedge \boldsymbol{a} \neq \boldsymbol{b}$.

This can be done efficiently, as the values of $\mathbf{dep}_{\mathrm{io}}$ for a given output actor a'_o are totally ordered under $\leq$, i.e., $\mathbf{dep}_{\mathrm{io}}(a'_o, n) \leq \mathbf{dep}_{\mathrm{io}}(a'_o, n+1)$.

Each input/output dependency state is a point in the n-dimensional Euclidean vector space ${\mathbb{N}_0}^{|A_\mathrm{I} \cup A_\mathrm{O}|}$ where each dimension represents the number of firings of a subgraph input or output actor, respectively. The initial state $\boldsymbol{q}_\perp$ is then trivially the all zero vector representing the fact that in the beginning no actor of the subgraph has fired.

As the calculation of the input/output dependency states Q_{io} considers each output actor individually, Q_{io} does not contain states resulting from different interleavings of actor firings that are permitted by the partial order of these actor firings. These interleavings are captured by the *state space* Q, which is defined as follows:

Definition 4.4 (*Cluster State Space*). The *state space* Q of a cluster is defined as the least fixpoint $Q = \mathbf{lfp}(Q' = \{\max(\boldsymbol{q}_1, \boldsymbol{q}_2) \mid \boldsymbol{q}_1, \boldsymbol{q}_2 \in Q'\} \cup Q' \cup Q_{io})$ which enlarges Q' starting from Q_{io} by adding the pointwise maximums of all pairs of input/output states from Q' until no more new states are created.

However, in order to avoid this explicit enumeration of the state space Q, *rules* are used to define valid state transitions implicitly. At a glance, a rule $r \in R$ maps a subspace of the vector space Q to a vector of actor firings $\boldsymbol{s}$ (in the following called *partial repetition vector*) that can be executed if the current state is contained in this subspace.

Note that we can define an equivalence relation between two states $\boldsymbol{q}'$ and $\boldsymbol{q}$ as follows: $\boldsymbol{q}' \equiv \boldsymbol{q} \iff \boldsymbol{q}' = \boldsymbol{q} + m \cdot \boldsymbol{\mu}_{io}, m \in \mathbb{Z}$, with $\boldsymbol{\mu}_{io} = (\mu_{a_{i,1}}, \mu_{a_{i,2}}, \ldots, \mu_{a_{i,|A_I|}}, \mu_{a_{o,1}}, \mu_{a_{o,2}}, \ldots, \mu_{a_{o,|A_\mathrm{O} - A_\mathrm{I}|}})$ being the permutation and projection of the repetition vector $\boldsymbol{\mu}_\mathrm{c}$ onto input and output actors in an order matching the definition of the input/output dependency states from Q_{io}. This equivalence is due to the fact that the state of the subgraph is the same before and after executing a schedule τ corresponding to the repetition vector $\boldsymbol{\mu}_\mathrm{c}$. For example, in Table I, equivalent input/output dependency states are amongst others $(0,1,0)$ and $(1,2,1)$ as well as $(2,1,0)$ and $(3,2,1)$.

Therefore, if a rule r is executed in state $\boldsymbol{q}_\mathrm{src}$ the destination state $\boldsymbol{q}_\mathrm{dst}$ is calculated in two steps: First, the actor firings specified by $\boldsymbol{s}$ are added to the current state, i.e., $\boldsymbol{q}'_\mathrm{dst} = \boldsymbol{q}_\mathrm{src} + \boldsymbol{s}$. Then, the new state $\boldsymbol{q}_\mathrm{dst}$ is the smallest equivalent state of $\boldsymbol{q}'_\mathrm{dst}$, i.e., $\boldsymbol{q}_\mathrm{dst} = \boldsymbol{q}'_\mathrm{dst} \bmod \boldsymbol{\mu}_{io}$.[5] More formally, a rule r is defined as follows:

Definition 4.5 (*Rule*). A rule is a tuple $r = (\boldsymbol{l}, \boldsymbol{u}, \boldsymbol{s})$. The vectors $\boldsymbol{l} \in {\mathbb{N}_0}^{|A_\mathrm{I} \cup A_\mathrm{O}|}$ and $\boldsymbol{u} \in {\mathbb{N}_{0,\infty}}^{|A_\mathrm{I} \cup A_\mathrm{O}|}$, $\boldsymbol{l} \leq \boldsymbol{u}$, define the subspace of the state space $Q_r \subseteq Q$ where the rule is active, i.e., $Q_r = \{\boldsymbol{q} \in Q \mid \boldsymbol{l} \leq \boldsymbol{q} \leq \boldsymbol{u}\}$.[6] The partial repetition vector $\boldsymbol{s} \in {\mathbb{N}_0}^{|A_\mathrm{I} \cup A_\mathrm{O}|}$ specifies for each input/output actor how many firings to perform when r is executed.

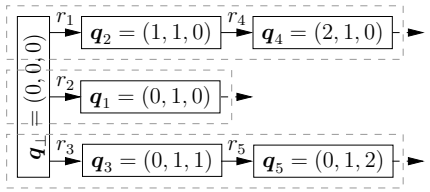
4.2. Derivation of Initial Rules

The next step of the proposed clustering approach is to find a set of *initial rules* R_ini based on the input/output dependency states Q_{io}. According to Section 5.2 Lemma 5.4, any set of rules with the property that all input/output dependency states are reachable via application of these rules starting from $\boldsymbol{q}_\perp$ is a valid set of initial rules. Note that this property has to be verified only for a subset of input/output dependency states due to the equivalence relation for input/output states defined above.

A rule r can be constructed from two input/output dependency states $\boldsymbol{q}_1 \in Q_{io}$ and $\boldsymbol{q}_2 \in Q_{io}$ with $\boldsymbol{q}_1 < \boldsymbol{q}_2$. Then, the rule r generated by these two states has to perform $r.\boldsymbol{s} = \boldsymbol{q}_2 - \boldsymbol{q}_1 > \boldsymbol{0}$ actor firings. The lower bound $r.\boldsymbol{l}$ is equal to $\boldsymbol{q}_1$, representing the

[5] The modulo operation between two vectors $\boldsymbol{a}'$ and $\boldsymbol{b}$ is defined such that $\boldsymbol{a}' \bmod \boldsymbol{b}$ denotes the smallest vector $\boldsymbol{a} = \boldsymbol{a}' - m \cdot \boldsymbol{b}, m \in \mathbb{N}_0$ under the constraint that $\boldsymbol{a} \geq \boldsymbol{0}$ still holds.

[6] $\mathbb{N}_{0,\infty} = \mathbb{N}_0 \cup \{\infty\}$ denotes the set of non-negative integers including infinity.



a) Input/output dependency state chains

r	$\boldsymbol{l}$	$\boldsymbol{u}$	$\boldsymbol{s}$
r_1	$(0,0,0)$	$(0,0,\infty)$	$(1,1,0)$
r_2	$(0,0,0)$	$(\infty,0,\infty)$	$(0,1,0)$
r_3	$(0,0,0)$	$(\infty,0,0)$	$(0,1,1)$
r_4	$(1,1,0)$	$(1,\infty,\infty)$	$(1,0,0)$
r_5	$(0,1,1)$	$(\infty,\infty,1)$	$(0,0,1)$

b) Initial rules R_{ini}

Fig. 3. a) Input/output dependency states for subgraph g_S from Fig. 1 (cf. Table I) partitioned into chains and annotated with the initial rules, and b) lower/upper bounds and partial repetition vector for the initial rules from Fig. 3a).

minimum number of actor firings which have to be performed in order to enable the rule. The upper bound $r.\boldsymbol{u}$ is calculated as follows:

$$\forall a : u_a = \left\{ \begin{array}{ll} l_a & \text{if } s_a > 0 \\ \infty & \text{otherwise} \end{array} \right\}$$

If an actor a is fired by r, i.e., $s_a > 0$, the upper bound is set to the lower bound. This constrains the number of firings of a to an interval which only contains the single value l_a. This requirement is due to the fact that we can only assume the availability of sufficient tokens in order to execute a for the lower bound l_a. If an actor a is not fired by r, we only require that the minimum number of actor firings have already been performed by setting $u_a = \infty$. This ensures that at least the minimum number of tokens have been produced by a. Note that if more firings of a have already been performed than indicated by l_a, a must also have produced more tokens than required, not less.

A valid set of initial rules is readily available by partitioning the partially ordered set of input/output dependency states into a set of chains $C_{io} = \{C_1, C_2, \ldots, C_n\}$[7]. Due to the equivalence relation for input/output states as defined above, the construction of chains can be stopped when all input/output states $q \not\geq \boldsymbol{\mu}_{io}$ are inserted into chains. Note that $\boldsymbol{q}_\perp$ must be added to each chain $C_k \in C_{io}$ to ensure the reachability of the smallest input/output state in the chain from $\boldsymbol{q}_\perp$.

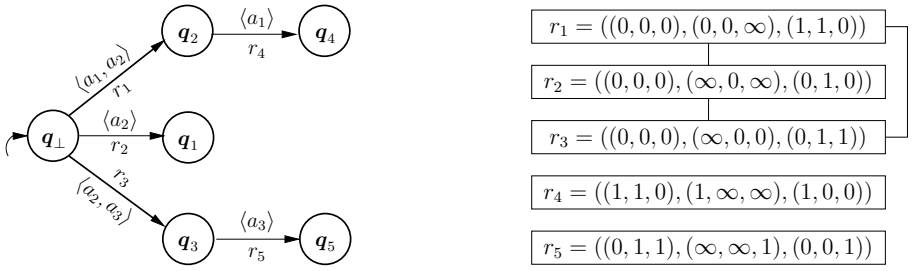
Then, for each chain $C_k \in C_{io}$ and each pair of states $(\boldsymbol{q}_1, \boldsymbol{q}_2)$ with $\boldsymbol{q}_1, \boldsymbol{q}_2 \in C_k, \boldsymbol{q}_1 < \boldsymbol{q}_2, \nexists \boldsymbol{q} \in C_k : \boldsymbol{q}_1 < \boldsymbol{q} < \boldsymbol{q}_2$, a rule is constructed as described above and added to the set of initial rules R_{ini}. The task of finding these state pairs can be efficiently accomplished if the chains are constructed appropriately.

As each chain is totally ordered, all states in a chain can be reached from $\boldsymbol{q}_\perp$ by application of rules from R_{ini} by construction. On the other hand, each input/output state $\boldsymbol{q} \not\geq \boldsymbol{\mu}_{io}$ has been added into exactly one chain. Therefore, all input/output dependency states are reachable via application of rules from R_{ini} starting from $\boldsymbol{q}_\perp$, i.e., R_{ini} is a valid set of initial rules.

Example 4.6. Considering again the dataflow graph given in Fig. 1, with the static dataflow subgraph g_S induced by $A_S = \{a_1, a_2, a_3\}$. The partitioning of input/output dependency states of g_S (cf. Table I) into chains is shown in Fig. 3a). The initial rule r_1 corresponding to edge $\boldsymbol{q}_\perp \rightarrow \boldsymbol{q}_2$ is then derived as follows: $\boldsymbol{s} = \boldsymbol{q}_2 - \boldsymbol{q}_\perp = (1,1,0)$, $\boldsymbol{l} = \boldsymbol{q}_\perp = (0,0,0)$, and $\boldsymbol{u} = (0,0,\infty)$. Therefore, in order to enable the rule, actors a_1 and a_2 must not have been fired before. Further initial rules are given in the table in Fig. 3b).

Simulating the initial rules R_{ini} given in Fig. 3 results in the FSM depicted in Fig. 4a). It can be observed that all states of the FSM shown in Fig. 2 are present. However, the

[7]A *chain* is a totally ordered subset C of a partially ordered set S, i.e., each pair of elements in C is comparable.



a) Simulated FSM b) Conflict graph for initial rules

Fig. 4. a) FSM resulting from simulating rules R_{ini} from Fig. 3. Applied rules are annotated to the edges, together with the actor firings performed by the rule. b) Conflict graph for the initial rules R_{ini}.

transitions are different, and not all states have outgoing transitions, therefore causing a deadlock if the FSM enters one of these states. These deadlocks are caused by so-called *conflict* rules: Two rules r_1 and r_2 are in *conflict* if they are both enabled in a given state s, but firing one prevents the firing of the other. In the example, both r_1 and r_3 are enabled in state $\boldsymbol{q}_\perp$. However, if r_1 is fired first, r_3 cannot be fired anymore. On the other hand, if r_3 is fired first, r_1 cannot be fired anymore. These *conflict* rules will be resolved in the following section.

4.3. Resolving Conflict Rules

Two rules are in conflict if both have at least one common state $\boldsymbol{q}$, and fire at least one common actor. Then, if the current state $\boldsymbol{q}_{cur} = \boldsymbol{q}$, both rules are enabled (assuming enough tokens are available). However, if r_1 is executed first, r_2 can no longer be applied, and vice versa.

Proof: Let a be the common actor which is fired by both rules. Then, by construction, the lower and upper bounds of the corresponding intervals are as follows: $r_1.l_a = r_1.u_a = n$ and $r_2.l_a = r_2.u_a = m$. Per assumption both rules have a common state. Hence, the intersection of their intervals is not empty. However, as both intervals contain only a single value, this can only be the case if $n = m$. Let $q_{cur,a} = n$ in order to enable both r_1 and r_2. Without loss of generality assume that r_1 is executed first. Then, for the next state, we have that $q_{next,a} = n + r_1.s_a$. Note that $r_1.s_a > 0$, as a is fired by r_1. However, as $q_{next,a} \neq m$ the rule r_2 is disabled. Analogously, if r_2 is fired first, r_1 will be disabled. □

Note that if two rules r_1 and r_2 have no common states, they cannot conflict, as they can never be enabled at the same time. Also, if two rules have indeed a common state $\boldsymbol{q}$ but do not fire common actors, they cannot conflict either: Without loss of generality, let a be an actor which is fired by r_1 but not by r_2. Then, by construction, the lower and upper bounds of the corresponding intervals are as follows: $r_1.l_a = r_1.u_a = n$ and $r_2.l_a = m, r_2.u_a = \infty$. Per assumption both rules have a common state. Hence, the intersection of their intervals is not empty. This can only be the case if $n \geq m$. Let $q_{cur,a} = n$ in order to enable both r_1 and r_2. If r_1 is fired first, it follows for the next state that $q_{next,a} = n + r_1.s_a > n \geq m$, i.e., r_2 is still enabled. If, on the other hand, r_2 is applied first, it follows for the next state that $q_{next,a} = n + r_2.s_a = n$, i.e., r_1 remains enabled.

In order to resolve the conflict situation, new rules will be added to R_{ini} by a least fixpoint operation to form R; the set of deadlock-free rules. For each conflicting rule pair r_1 and r_2, common actor firings are extracted from both rules, resulting in two additional rules r_1^1 and r_2^1 in the general case. Then, r_2^1 can be applied after r_1, and analogously, r_1^1 can be executed after r_2 (For a detailed proof cf. Section 5.2 Lemma 5.3). More formally, the conflict resolving operation is defined as follows:

Definition 4.7 (*Conflict Resolving Operation*). Given two conflicting rules r_1 and r_2, the conflict resolving operation $\diamond$ maps these rules to two new rules r_1^1 and r_2^1 such that r_2^1 can be executed after r_1, and r_1^1 can be executed after r_2, i.e., $(r_1^1, r_2^1) = r_1 \diamond r_2$. Let $\boldsymbol{s}_c = \min(r_1.\boldsymbol{s}, r_2.\boldsymbol{s})$ be the pointwise minimum of the partial repetition vectors $r_1.\boldsymbol{s}$ and $r_2.\boldsymbol{s}$. Note that $\boldsymbol{s}_c$ represents the common actor firings of r_1 and r_2. Then, the lower bounds and partial repetition vectors of r_1^1 and r_2^1 can be calculated as follows:

$$r_1^1.\boldsymbol{l} = r_1.\boldsymbol{l} + \boldsymbol{s}_c \qquad r_1^1.\boldsymbol{s} = r_1.\boldsymbol{s} - \boldsymbol{s}_c$$
$$r_2^1.\boldsymbol{l} = r_2.\boldsymbol{l} + \boldsymbol{s}_c \qquad r_2^1.\boldsymbol{s} = r_2.\boldsymbol{s} - \boldsymbol{s}_c$$

The upper bounds $r_1^1.\boldsymbol{u}$ and $r_2^1.\boldsymbol{u}$ are then calculated from these lower bounds and partial repetition vectors as described in Section 4.2.

Finally, the set of deadlock-free rules R is calculated by the least fixpoint operation $R = \mathbf{lfp}(R' = \{r_1^1, r_2^1 \mid r_1, r_2 \in R' \wedge r_1 \text{ and } r_2 \text{ are in conflict}, (r_1^1, r_2^1) = r_1 \diamond r_2\} \cup R' \cup R_{\text{ini}})$ which enlarges R' starting from R_{ini} by adding the rules r_1^1 and r_2^1 calculated by the conflict resolving operation $\diamond$ from the two conflicting rules r_1 and r_2 until no more new rules are created.

A special case, which leads to the elimination of *redundant* rules during conflict resolving arises, if for a pair of conflicting rules r_1 and r_2 one rule is more general than the other one. A rule r_1 is more general than rule r_2 if the set of states Q_1 where r_1 can be applied is a superset of the set of states Q_2 where r_2 can be applied, i.e., $r_1.\boldsymbol{l} \leq r_2.\boldsymbol{l} \wedge r_1.\boldsymbol{u} \geq r_2.\boldsymbol{u}$, and r_1 fires less actors than r_2, i.e., $r_1.\boldsymbol{s} < r_2.\boldsymbol{s}$.

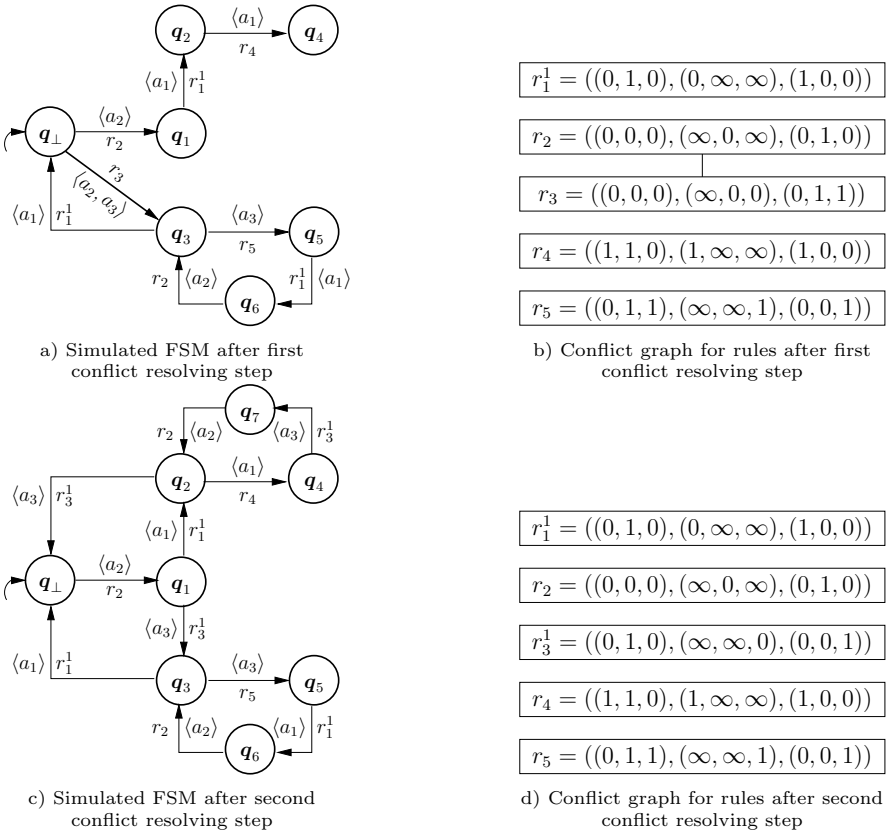
Conflict resolution will generate rules r_1^1 and r_2^1. In this case, the common actor firings are $\boldsymbol{s}_c = \min(r_1.\boldsymbol{s}, r_2.\boldsymbol{s}) = r_1.\boldsymbol{s}$, and thus, $r_1^1.\boldsymbol{s} = \boldsymbol{0}$. This means that r_1^1 can be discarded, as it does not fire any actors. Also, r_2 can be eliminated as a redundant rule.

Proof: r_2^1 can be executed after r_1 and will lead to the same state as firing r_2 directly (For proof cf. Section 5 Lemma 5.3). Consequently, in all states where r_2 can be executed, the sequence of rule applications r_1 and r_2^1 can also be executed, and will result in the same destination state. Hence, rule r_2 is redundant. □

Example 4.8. Considering the set of initial rules $R_{\text{ini}} = \{r_1, r_2, \ldots r_5\}$ from Fig. 3, the corresponding conflict graph is shown in Fig. 4b). In the first step, the conflict between r_1 and r_2 will be resolved. A common state of these rules is, e.g., $\boldsymbol{q}_\perp$. The vector of common actor firings is $\boldsymbol{s}_c = (0, 1, 0)$. Hence, the conflict resolving operation $r_1 \diamond r_2$ results in two rules: $r_1^1 = ((0,1,0), (0,\infty,\infty), (1,0,0))$, and $r_2^1 = ((0,1,0), (\infty,\infty,\infty), (0,0,0))$ (which can be discarded due to the zero partial repetition vector). Note that r_2 is more general than r_1. Therefore, r_1 can be eliminated as redundant rule. It should be noted that this conflict resolving operation also resolves the conflict between r_1 and r_3. The set of rules after this first conflict resolving step is $R_{\text{ini}}^1 = \{r_1^1, r_2, r_3, r_4, r_5\}$. The resulting FSM and conflict graph are depicted in Fig. 5a) and b). Analogously, the second step resolves the conflict between r_2 and r_3. The deadlock-free set of rules after this second (and last) conflict resolving step is $R = \{r_1^1, r_2, r_3^1, r_4, r_5\}$. The resulting FSM and conflict graph are depicted in Fig. 5c) and d).

Comparing the FSM in Fig. 2 to the FSM obtained by simulation in Fig. 5c), the former features a transition from $\boldsymbol{q}_4$ to $\boldsymbol{q}_2$ firing both a_2 and a_3, while in the latter, this transition is split into two transitions: One transition from $\boldsymbol{q}_4$ to $\boldsymbol{q}_7$ firing a_3 only, and a second transition from $\boldsymbol{q}_7$ to $\boldsymbol{q}_2$ firing a_2. The reason for this can be found in the conflict resolution step: Although it generates rules which guarantee a deadlock-free execution of the composite actor, it may also generate rules which do not fire any input actors, but only output actors.

While this is perfectly valid if counter variables for both input and output actors are maintained during the execution of the composite actor, it becomes problematic if only counter variables for input actors are used, which is desirable, as it means less checks on



a) Simulated FSM after first conflict resolving step

b) Conflict graph for rules after first conflict resolving step

c) Simulated FSM after second conflict resolving step

d) Conflict graph for rules after second conflict resolving step

Fig. 5. a) FSM resulting from simulating rules R^1_{ini} after the first conflict resolving step. b) Conflict graph for the rules R^1_{ini}. c) FSM resulting from simulating the deadlock-free rules R after the second conflict resolving step. d) Conflict graph for the rules R.

rule conditions, and thus, less scheduling overhead for the composite actor. This is possible, as the state of the cluster can always be determined solely based on the number of input actor firings.

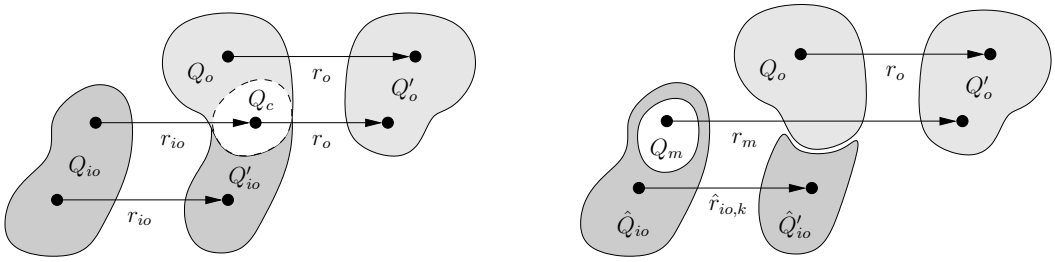
For example, consider rule r_2 in Fig. 5d), which is only applicable if the counter for output actor a_2 is zero. However, eliminating the condition for a_2 results in a rule which is always applicable, as the firing intervals corresponding to a_1 and a_3 are not constrained.

In order to eliminate the counter variables for the output actors, we must first eliminate the rules which do fire only output actors. This is discussed next.

4.4. Rule Merging

The firing interval boundary vectors $r_k.\boldsymbol{l}$ and $r_k.\boldsymbol{u}$ of a rule r_k represent the set of states Q_k in which r_k can be applied. Therefore, an *image* operation can be defined which calculates the set of states Q'_k (also in the form of interval boundary vectors) after r_k has been executed, by simply adding the partial repetition vector $r_k.\boldsymbol{s}$ to $r_k.\boldsymbol{l}$ and $r_k.\boldsymbol{u}$. Analogously, a *pre-image* operation can be defined which reverses the image operation by subtracting the partial repetition vector $r_k.\boldsymbol{s}$ from two interval boundary vectors $\boldsymbol{l}$ and $\boldsymbol{u}$.

Let r_{io} be a rule which fires at least one input actor (and possibly some output actors), and r_o be a rule which fires only output actors. In principle, r_o can be applied after r_{io} if the set of states $Q_c = Q'_{io} \cap Q_o$ is non-empty, i.e., if r_o has some common states with



a) Rules r_{io} and r_o before merging

b) Rules r_m and $\hat{r}_{io,k}$ after merging

Fig. 6. Rule merging operation: a) Rules r_{io} and r_o are merged, b) resulting in rules r_m and at least one rule $\hat{r}_{io,k}$.

Q'_{io}. Note that in this case, the firing interval boundary vectors $\boldsymbol{l}_c$ and $\boldsymbol{u}_c$ corresponding to Q_c can be calculated by intersecting the firing condition intervals of Q_o and Q'_{io}, i.e., $\boldsymbol{l}_c = \max(r_o.\boldsymbol{l}, r_{io}.\boldsymbol{l} + r_{io}.\boldsymbol{s})$, and $\boldsymbol{u}_c = \min(r_o.\boldsymbol{u}, r_{io}.\boldsymbol{u} + r_{io}.\boldsymbol{s})$.

Then, r_{io} can be merged with r_o, resulting in a new rule r_m, which combines the actor firings performed separately by r_{io} and r_o, i.e., $r_m.\boldsymbol{s} = r_{io}.\boldsymbol{s} + r_o.\boldsymbol{s}$ (cf. Fig. 6). The firing interval boundary vectors of r_m can be calculated as the pre-image of Q_c w.r.t. the partial repetition vector of r_{io}, i.e, $r_m.\boldsymbol{l} = \boldsymbol{l}_c - r_{io}.\boldsymbol{s}$, and $r_m.\boldsymbol{u} = \boldsymbol{u}_c - r_{io}.\boldsymbol{s}$.

Note that $r_m.\boldsymbol{l} = \boldsymbol{l}_c - r_{io}.\boldsymbol{s} = \max(r_o.\boldsymbol{l}, r_{io}.\boldsymbol{l} + r_{io}.\boldsymbol{s}) - r_{io}.\boldsymbol{s} \geq r_{io}.\boldsymbol{l} + r_{io}.\boldsymbol{s} - r_{io}.\boldsymbol{s} = r_{io}.\boldsymbol{l}$, i.e., $r_m.\boldsymbol{l} \geq r_{io}.\boldsymbol{l}$. Also, $r_m.\boldsymbol{u} = \boldsymbol{u}_c - r_{io}.\boldsymbol{s} = \min(r_o.\boldsymbol{u}, r_{io}.\boldsymbol{u} + r_{io}.\boldsymbol{s}) - r_{io}.\boldsymbol{s} \leq r_{io}.\boldsymbol{u} + r_{io}.\boldsymbol{s} - r_{io}.\boldsymbol{s} = r_{io}.\boldsymbol{u}$, i.e., $r_m.\boldsymbol{u} \leq r_{io}.\boldsymbol{u}$. This means that $Q_m \subseteq Q_{io}$, as shown in Fig. 6b).

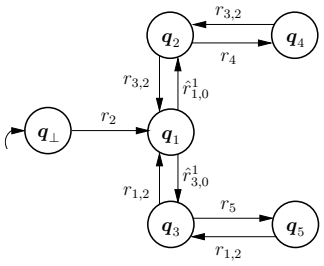
However, inserting such a merged rule r_m into the conflict-free set of rules would destroy this property, as r_{io} conflicts with r_m due to common actor firings ($\min(r_m.\boldsymbol{s}, r_{io}.\boldsymbol{s}) = \min(r_{io}.\boldsymbol{s} + r_o.\boldsymbol{s}, r_{io}.\boldsymbol{s}) = r_{io}.\boldsymbol{s}$) and common states ($Q_m \subseteq Q_{io}$ shown above). Therefore, the merge operation must also *restrict* the firing condition intervals of the original rule r_{io} such that it can no longer be applied to states in Q_m (where r_m can be applied). To this end, the set $\hat{Q}_{io} = Q_{io} \setminus Q_m$ has to be calculated in terms of interval boundary vectors. Without going into detail, this operation results in a set of rules $\hat{R}_{io} = \{\hat{r}_{io,0}, \hat{r}_{io,1}, \ldots, \hat{r}_{io,n}\}$ necessary to cover all states in $\hat{Q}_{io}$. The rules in $\hat{R}_{io}$ may not be conflict-free, but as they are derived from the same rule, namely r_{io}, they can be safely added to the set of rules R in this case. Note that as r_o will be eliminated later, it doesn't have to be restricted like r_{io}.

An important point which has to be considered during the rule merging step is that if r_o is eliminated after the rule merging step, the merge operation has to be an over-approximation. Otherwise, some states in the FSM may not have outgoing transitions, therefore causing deadlocks during the execution of the composite actor. Given two rules r_{io} and r_o, Q_c may be empty, in which case they won't be merged. However, analogous to input/output dependency states, we can also define an equivalence relation between two rules r and r' as follows: $r \equiv r' \iff r.\boldsymbol{s} = r'.\boldsymbol{s} \wedge r.\boldsymbol{l} = r'.\boldsymbol{l} + m \cdot \boldsymbol{\mu}_{io} \wedge r.\boldsymbol{u} = r'.\boldsymbol{u} + m \cdot \boldsymbol{\mu}_{io}, m \in \mathbb{Z}$. As a result, when merging rules, we must also consider equivalent rules of r_o, i.e., rules $r_{o,m} = (r_o.\boldsymbol{l} + m \cdot \boldsymbol{\mu}_{io}, r_o.\boldsymbol{u} + m \cdot \boldsymbol{\mu}_{io}, r_o.\boldsymbol{s})$ with $m > 0$. If, however, due to $m > 0$, the merged rule r_m has $r_m.\boldsymbol{l} \geq \boldsymbol{\mu}_{io}$, it can safely be discarded, as the counter variables maintained during simulation of the composite actor cannot reach the values of such lower bounds of the condition intervals.

Finally, a special rule may exist which fires only output actors, but must not be eliminated after the rule merging step: This is the case if sufficient initial tokens are contained in the cluster such that output actors can be fired without firing any input actors. Obviously, eliminating this special rule would prevent the composite actor from running at all, as the initial state $\boldsymbol{q}_\perp$ would have no outgoing transitions. As only one such rule can exist, it can easily be tagged as special and thus can be retained after the merging step.

r	$\boldsymbol{l}$	$\boldsymbol{u}$	$\boldsymbol{s}$
$\hat{r}^1_{1,0}$	$(0,1,0)$	$(0,\infty,0)$	$(1,0,0)$
$r_{1,2}$	$(0,1,1)$	$(0,1,\infty)$	$(1,1,0)$
r_2	$(0,0,0)$	$(1,0,1)$	$(0,1,0)$
$\hat{r}^1_{3,0}$	$(0,1,0)$	$(0,\infty,0)$	$(0,0,1)$
$r_{3,2}$	$(1,1,0)$	$(\infty,1,0)$	$(0,1,1)$
r_4	$(1,1,0)$	$(1,\infty,\infty)$	$(1,0,0)$
r_5	$(0,1,1)$	$(\infty,\infty,1)$	$(0,0,1)$

a) Final rules R_{fin}



b) Simulated final rules R_{fin}

Fig. 7. a) Final rules R_{fin} for subgraph g_S from Fig. 1 derived from the conflict-free rules in Fig. 5d) via merging, and b) FSM resulting from simulating rules R_{fin}. Applied rules are annotated to the edges.

Example 4.9. Consider the conflict-free set of rules $R = \{r^1_1, r_2, r^1_3, r_4, r_5\}$ from Fig. 5d). Rule r_2 fires only output actors, namely a_2. First, we will merge it with rule r^1_1, which fires a_1, an input/output actor. For $m = 0$, $\boldsymbol{l}_c = \max((0,0,0),(0,1,0)+(1,0,0)) = (1,1,0)$, and $\boldsymbol{u}_c = \min((\infty,0,\infty),(0,\infty,\infty)+(1,0,0)) = (0,1,\infty)$. In this case, we cannot merge r^1_1 and $r_{2,0}$, as $\boldsymbol{l}_c \not\leq \boldsymbol{u}_c$.

However, for $m = 1$, $\boldsymbol{l}_c = \max((0,0,0)+(1,1,1),(0,1,0)+(1,0,0)) = (1,1,1)$ and $\boldsymbol{u}_c = \min((\infty,0,\infty)+(1,1,1),(0,\infty,\infty)+(1,0,0)) = (1,1,\infty)$. Now, $\boldsymbol{l}_c \leq \boldsymbol{u}_c$, and we can merge r^1_1 and $r_{2,1}$. The merged rule is calculated as $r_{1,2} = ((1,1,1)-(1,0,0),(1,1,\infty)-(1,0,0),(1,0,0)+(0,1,0)) = ((0,1,1),(0,1,\infty),(1,1,0))$. The set of restricted rules $\hat{R}^1_1$ contains only a single rule (after pruning another rule which could never be applied), namely $\hat{r}^1_{1,0} = ((0,1,0),(0,\infty,0),(1,0,0))$.

It can be observed that $r_{1,2}$ can be applied in state $\boldsymbol{q}_5 = (0,1,2)$ (cf. Fig. 5c)), creating the missing (direct) transition from $\boldsymbol{q}_5$ to $\boldsymbol{q}_3$ (cf. Fig. 2). Moreover, the restricted rule r^1_1 in the form of $\hat{r}^1_{1,0}$ can no longer be applied in $\boldsymbol{q}_5$, thus eliminating state $\boldsymbol{q}_6$.

Merging r_2 with the remaining states r^1_3, r_4 and r_5 results in rules R_{fin} given in Fig. 7b). Note that r_2 has been retained as discussed above.

The next step of the proposed clustering approach is to schedule the partial repetition vector of each rule $r \in R_{\text{fin}}$, the set of rules after the merging step.

4.5. Scheduling Static Sequences

The partial repetition vectors $r.\boldsymbol{s}$ are scheduled by a modified version of the cycle-breaking algorithm presented in [Hsu and Bhattacharyya 2007], which, for pure SDF graphs always finds a single appearance schedule (SAS) if one exists. In a SAS, each actor occurs only once, therefore minimizing code memory size when inlining the actor firings.

Basically, a SAS corresponds to a topological sorting of the actors contained in the static dataflow subgraph. This is achieved by topologically sorting the strongly connected components (SCC) of the subgraph. Subsequently, for each non-trivial SCC, the algorithm removes the edges with enough initial tokens to perform a whole iteration of the static subgraph induced by the SCC. If the SCC is still strongly connected after removing such edges, the subgraph is said to be *tightly* connected and is scheduled by simulation. Otherwise, the SCC is *loosely* connected, and the algorithm is applied recursively to this SCC.

For partial repetition vectors, however, it may not always be possible to find a SAS even if one exists. This is due to the fact that the partial repetition vector in question may not be a multiple of the repetition vector of the static subgraph induced by the SCC.

The static sequences are used as basis for software synthesis [Bhattacharyya et al. 1988] into C++ source code which is itself compiled by gcc to generate the executables.

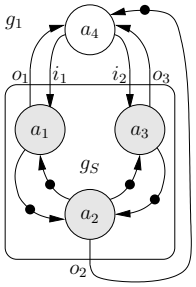


Fig. 8. The dataflow graph from Figure 1 with port annotations as needed for the denotational descriptions via KPN.

4.6. Clustering Condition

The proposed clustering approach is based on an implicit state space representation. However, the viability of this approach depends on the input/output dependency function (cf. Definition 4.1) to be periodic with respect to the repetition vector $\boldsymbol{\mu}$. This is the case iff for a sufficiently large (but finite) n, $\forall a_i \in A_I : \mathbf{dep}_{\mathrm{io}}(a_o, n) \geq \mu_{a_i}$, i.e., after enough firings of a given output actor, the number of input actor firings is greater or equal to the corresponding values of the repetition vector. Otherwise, there is an infinite number of non-equivalent states $\boldsymbol{q} \ngeq \boldsymbol{\mu}_{io}$ in the input/output state set and as a consequence the QSS can not be constructed. A necessary and sufficient condition in order to guarantee this property is the *clustering condition*:

Definition 4.10 (*Clustering Condition*). A dataflow subgraph g_{S} can be clustered by the proposed method if the subgraph only contains static actors, the subgraph disregarding its inputs and outputs is deadlock free itself, and for each pair of subgraph input and output actors $(a_{\mathrm{i}}, a_{\mathrm{o}}) \in A_{\mathrm{I}} \times A_{\mathrm{O}}$ there exists a directed path $p \in C^*$ from actor a_{i} to actor a_{o}.

For static dataflow subgraphs g_{S} that do not satisfy the cluster condition, the set of static actors A_{S} can always be partitioned into subsets, each of which induces a connected subgraph satisfying the *clustering condition*. This partitioning can be performed by consecutively removing edges with the unbounded token accumulation problem from the subgraph g_{S}. This operation will result in a decomposition of the subgraph g_{S} into m connected subgraphs satisfying the clustering condition.

5. PROVING SEMANTIC EQUIVALENCE

Before presenting experimental results showing the benefits of the presented clustering algorithm, a formal proof of the correctness of the approach is given.

The semantic equivalence of the composite actor a_{c} and the static dataflow graph g_{S} is proven in Theorem 5.5 by showing the equivalence of denotational descriptions ($F_{a_{\mathrm{c}}}$ and $F_{g_{\mathrm{S}}}$) via *Kahn process networks (KPN)* [Kahn 1974] of both a_{c} and g_{S}. To exemplify the denotational description, the dataflow graph from Figure 1 is used. For the sake of clarity, this dataflow graph is replicated with port annotations in Figure 8. Its denotational description $F_{g_{\mathrm{S}}}$ maps tuples $\boldsymbol{\nu}_{\mathrm{in}} = (\nu_{i_1}, \nu_{i_2})$ of sequences $\nu \in V^{**}$ of tokens $\bullet$ on its input ports i_1 and i_2 to tuples $F_{g_{\mathrm{S}}}(\boldsymbol{\nu}_{\mathrm{in}}) = \boldsymbol{\nu}_{\mathrm{out}} = (\nu_{o_1}, \nu_{o_2}, \nu_{o_3})$ of sequences of tokens on its output ports o_1, o_2, and o_3.[8] As can be seen from the example values of $F_{g_{\mathrm{S}}}$ given below, the denotational description naturally expresses the maximum output from input property, which should be preserved by the clustering algorithm.

[8]We use $\mathcal{V}^{**}$ to denote the set of all possible *finite* and *infinite sequences* of tokens $\bullet \in \mathcal{V}$, i.e., $\mathcal{V}^{**} = \bigcup_{n \in \{0,1,\ldots\infty\}} \mathcal{V}^n$

$$F_{g_S}((),()) = ((),(\bullet),())$$

$$\begin{aligned} F_{g_S}((\bullet),()) &= ((\bullet),(\bullet),()) & F_{g_S}((),(\bullet)) &= ((),(\bullet),(\bullet)) \\ F_{g_S}((\bullet,\bullet),()) &= ((\bullet,\bullet),(\bullet),()) & F_{g_S}((),(\bullet,\bullet)) &= ((),(\bullet),(\bullet,\bullet)) \\ F_{g_S}((\bullet,\bullet,\ldots),()) &= ((\bullet,\bullet),(\bullet),()) & F_{g_S}((),(\bullet,\bullet,\ldots)) &= ((),(\bullet),(\bullet,\bullet)) \end{aligned}$$

$$F_{g_S}((\bullet),(\bullet)) = ((\bullet),(\bullet,\bullet),(\bullet))$$

$$\begin{aligned} F_{g_S}((\bullet,\bullet),(\bullet)) &= ((\bullet,\bullet),(\bullet,\bullet),(\bullet)) & F_{g_S}((\bullet),(\bullet,\bullet)) &= ((\bullet),(\bullet,\bullet),(\bullet,\bullet)) \\ F_{g_S}((\bullet,\bullet,\bullet),(\bullet)) &= ((\bullet,\bullet,\bullet),(\bullet,\bullet),(\bullet)) & F_{g_S}((\bullet),(\bullet,\bullet,\bullet)) &= ((\bullet),(\bullet,\bullet),(\bullet,\bullet,\bullet)) \\ F_{g_S}((\bullet,\bullet,\bullet,\ldots),(\bullet)) &= ((\bullet,\bullet,\bullet),(\bullet,\bullet),(\bullet)) & F_{g_S}((\bullet),(\bullet,\bullet,\bullet,\ldots)) &= ((\bullet),(\bullet,\bullet),(\bullet,\bullet,\bullet)) \end{aligned}$$

$$\ldots$$

Note that the token values have been abstracted away by using $\bullet$ as token symbol. This can be done because clustering can also be thought of as KPN process composition. Therefore, the same operations will always be performed on the same values. However, it is not trivially obvious if the set of rules generated by the clustering algorithm will always produce the maximum number of output tokens from the given input tokens. For example, a fully static scheduling of g_S by the composite actor a_c would produce the following KPN function F_{a_c}, which is not equivalent to F_{g_S}:

$$F_{a_c}((),()) = ((),(),())$$

$$\begin{aligned} F_{a_c}((\bullet),()) &= ((),(),()) & F_{a_c}((),(\bullet)) &= ((),(),()) \\ F_{a_c}((\bullet,\ldots),()) &= ((),(),()) & F_{a_c}((),(\bullet,\ldots)) &= ((),(),()) \end{aligned}$$

$$F_{a_c}((\bullet),(\bullet)) = ((\bullet),(\bullet),(\bullet))$$

$$F_{a_c}((\bullet,\ldots),(\bullet)) = ((\bullet),(\bullet),(\bullet)) \quad F_{a_c}((\bullet),(\bullet,\ldots)) = ((\bullet),(\bullet),(\bullet))$$

$$\ldots$$

Therefore, by proving semantic equivalence we also prove that the clustering algorithm is correct. The opposite, that a correct (in the sense of always producing maximum output from minimal input) clustering algorithm will produce a semantically equivalent F_{a_c} from F_{g_S} is trivially true.

Before tackling the correctness proof for the clustering algorithm some preparation is required. First the semantic equivalence proof requires that for each pair of states $\boldsymbol{q}_x, \boldsymbol{q}_y \in Q$ with $\boldsymbol{q}_y \geq \boldsymbol{q}_x$ a valid schedule ν to change from $\boldsymbol{q}_x$ to $\boldsymbol{q}_y$ exists. The proof for the existence of ν is derived from the *max state reachability* lemma given below.

5.1. Max State Reachability

The notation $\boldsymbol{q}_x \xrightarrow{\nu} \boldsymbol{q}_y$ is used to denote the existence of a valid schedule ν, which can be executed in state $\boldsymbol{q}_x$ leading to state $\boldsymbol{q}_y$. Furthermore, a state $\boldsymbol{q}$ is called *reachable* iff $\boldsymbol{q}_\perp \xrightarrow{\nu} \boldsymbol{q}$ and a state $\boldsymbol{q}_y$ is called *reachable from* $\boldsymbol{q}_x$ iff $\boldsymbol{q}_x \xrightarrow{\nu} \boldsymbol{q}_y$. In the following the reachability of state $\boldsymbol{q}_m = \max(\boldsymbol{q}_x, \boldsymbol{q}_y)$ from both states $\boldsymbol{q}_x$ and $\boldsymbol{q}_y$ is proven under the precondition that these two states are reachable themselves.

LEMMA 5.1 (MAX STATE REACHABILITY). *If two states* $\boldsymbol{q}_x$, $\boldsymbol{q}_y \in Q$ *are* reachable *the pointwise maximum of these two states* $\boldsymbol{q}_m = \max(\boldsymbol{q}_x, \boldsymbol{q}_y)$ *is* reachable *from both* $\boldsymbol{q}_x$ *and* $\boldsymbol{q}_y$.

Proof: The proof is based on the well known *freedom of conflict property* of SDF systems. The property can be derived by considering SDF as a subclass of place/transition Petri nets (p/t-nets, for short), where actors correspond to transitions and channels to places. In general p/t-nets, firing a transition can disable another transition which was previously enabled. This situation is referred to as a conflict. However for the subclass of p/t-nets corresponding to SDF this can never happen [Murata 1989]. Therefore, for SDF systems, an actor a, once enabled, cannot be disabled by firing another actor $a' \in A \setminus \{a\}$.

Without loss of generality only the existence of $\boldsymbol{q}_x \xrightarrow{\nu_{x,m}} \boldsymbol{q}_m$ is proven. Given two reachable states $\boldsymbol{q}_x$ ($\boldsymbol{q}_\perp \xrightarrow{\nu_x} \boldsymbol{q}_x$) and $\boldsymbol{q}_y$ ($\boldsymbol{q}_\perp \xrightarrow{\nu_y} \boldsymbol{q}_y$) and let A' be the set of actors which have more occurrences in ν_y than in ν_x, i.e., $A' = \{a \in A \mid \#_a\nu_y > \#_a\nu_x\}$.[9] If A' is empty the proposition $\boldsymbol{q}_x \xrightarrow{\nu_{x,m}} \boldsymbol{q}_m$ is trivially true as $\boldsymbol{q}_m = \boldsymbol{q}_x$. Otherwise, let ν'_y be the longest prefix of ν_y with the property that each actor $a \in A'$ occurs less than or equal to $\#_a\nu_x$. Furthermore, due to $\forall a \in A \setminus A' : \#_a\nu'_y \leq \#_a\nu_y \leq \#_a\nu_x$ all actors $a \in A$ occur less than or equal to $\#_a\nu_x$ in schedule ν'_y. Moreover, after executing schedule ν'_y at least one enabled actor $a' \in A'$ exists. Therefore, due to the SDF freedom of conflict property this actor a' must also be enabled after executing schedule ν_x. Firing a' in state $\boldsymbol{q}_x$ will lead to a new state $\boldsymbol{q}'_x$ with property $\boldsymbol{q}_m = \max(\boldsymbol{q}'_x, \boldsymbol{q}_y)$. Furthermore, the existence of a $\boldsymbol{q}'_x \xrightarrow{\nu_{x',m}} \boldsymbol{q}_m$ is proven by induction where $A' = \emptyset$ is the induction start. Finally, $\nu_{x,m}$ is given by concatenating the schedules $\langle a' \rangle$ and $\nu_{x',m}$. □

Next the proof of the existence of a valid schedule $\boldsymbol{q}_x \xrightarrow{\nu} \boldsymbol{q}_y$ for all $\boldsymbol{q}_y \geq \boldsymbol{q}_x$ with $\boldsymbol{q}_x, \boldsymbol{q}_x \in Q$ is sketched using the *max state reachability* lemma. This lemma requires the reachability of the states $\boldsymbol{q}_x$ and $\boldsymbol{q}_y$. First note that all states $\boldsymbol{q} \in Q_{io}$ are trivially reachable by construction, i.e., the definition of the *input/output dependency function* (cf. Definition 4.1) implies a valid schedule ν and the *input/output dependency states* are basically a subset of the input/output dependency function values where the number of output actor firings have been maximized with respect to the available input actor firings. The remaining states $Q \setminus Q_{io}$ are derived by pairwise maximum operations from Q_{io} via a least fixpoint operation (cf. Definition 4.4). Let $\boldsymbol{q}_m = \max(\boldsymbol{q}_x, \boldsymbol{q}_y)$ be a state, which is added by a step of the lfp operation, then $\boldsymbol{q}_\perp \xrightarrow{\nu_x \frown \nu_{x,m}} \boldsymbol{q}_m$ is a schedule which reaches $\boldsymbol{q}_m$.[10]

With the above reasoning it is proven that all states $\boldsymbol{q} \in Q$ are reachable and that $\forall \boldsymbol{q}_x, \boldsymbol{q}_y \in Q : \boldsymbol{q}_y \geq \boldsymbol{q}_x \implies \boldsymbol{q}_x \xrightarrow{\nu_{x,y}} \boldsymbol{q}_y$ exists. However, the existence of a sequence $s_{x,y}$ of rule applications realizing the schedule $\nu_{x,y}$ also needs to be proven.

5.2. Max State Reachability via Rules

First some more notation is required: $\boldsymbol{q}_x \xrightarrow{r} \boldsymbol{q}_y$ denotes the existence of a rule r which can be applied in state $\boldsymbol{q}_x$ leading to a new state $\boldsymbol{q}_y$, while $\boldsymbol{q}_x \xrightarrow{s_{x,y}} \boldsymbol{q}_y$ denotes the existence of a sequence $s_{x,y}$ of rule applications from state $\boldsymbol{q}_x$ to state $\boldsymbol{q}_y$, i.e., $\boldsymbol{q}_x = \boldsymbol{q}_{x,0} \xrightarrow{r_{x,1}} \boldsymbol{q}_{x,1} \xrightarrow{r_{x,2}} \ldots \xrightarrow{r_{x,i}} \boldsymbol{q}_{x,i} = \boldsymbol{q}_y$. Furthermore, a state $\boldsymbol{q}$ is called *rule reachable* iff $\boldsymbol{q}_\perp \xrightarrow{s_x} \boldsymbol{q}$ and a state $\boldsymbol{q}_y$ is called *rule reachable from* $\boldsymbol{q}_x$ iff $\boldsymbol{q}_x \xrightarrow{s_{x,y}} \boldsymbol{q}_y$. Moreover, rules r are not of an arbitrary form but obey the *rule property* as defined below to be well formed:

Definition 5.2 (*Rule Property*). A rule r obeys the rule property iff $r.s_a = 0$ then $r.u_a = \infty$ and iff $r.s_a \neq 0$ then $r.l_a = r.u_a$.

[9] The notation $\#_a\nu$ is used to denote the number of occurrences of actor a in the schedule ν, e.g., $\#_{a_x}\langle a_x, a_y, a_x \rangle = 2$.

[10] The '$\frown$'-operator is used to denote the concatenation of sequences, e.g., for the two schedules $\nu_x = \langle a_y, a_y, a_3 \rangle$ and $\nu_y = \langle a_4, a_4 \rangle$ the concatenation equals $\nu_x \frown \nu_y = \langle a_y, a_y, a_3, a_4, a_4 \rangle$.

In other words, for a rule r to be well formed, actors which are not fired by the rule r may have an arbitrary amount of firings, as long as the lower bound $r.l_a$ is satisfied, while actors which are fired by rule r must have a concrete number of firings $r.l_a = r.u_a$. This is trivially true for the *initial rules* as described in Section 4.2 as well as rules derived by conflict resolution in Definition 4.7.

In the following, the rule reachability of $\boldsymbol{q}_m = \max(\boldsymbol{q}_x, \boldsymbol{q}_y)$ from both states $\boldsymbol{q}_x$ and $\boldsymbol{q}_y$ is proven, assuming all rules to be well formed. First a special case is proven in Lemma 5.3 for rules r_x and r_y which are enabled in a common state $\boldsymbol{q}$. For this limited case, the sequences consist both of a single rule, i.e., $s_{x,m} = \boldsymbol{q}_x \xrightarrow{r_y^1} \boldsymbol{q}_m$ and $s_{y,m} = \boldsymbol{q}_y \xrightarrow{r_x^1} \boldsymbol{q}_m$, which execute the schedule $\nu_{x,m}$ and $\nu_{y,m}$, respectively. Later on, this result will be extended by Lemma 5.4 to all rule reachable states $\boldsymbol{q}_x, \boldsymbol{q}_y \in Q$. From this result a proof is sketched that all states $\boldsymbol{q} \in Q$ are rule reachable. More formally the limited case is given below:

Lemma 5.3 (Single-Step Max State Rule Reachability). *Given two rules $r_x, r_y \in R$, which can both be applied in state $\boldsymbol{q}$, i.e., $r_x.\boldsymbol{l} \leq \boldsymbol{q} \leq r_x.\boldsymbol{u}$ and $r_y.\boldsymbol{l} \leq \boldsymbol{q} \leq r_y.\boldsymbol{u}$. Rule application of r_x or r_y leading to the state $\boldsymbol{q} \xrightarrow{r_x} \boldsymbol{q} + r_x.\boldsymbol{s} = \boldsymbol{q}_x$ or $\boldsymbol{q} \xrightarrow{r_y} \boldsymbol{q} + r_y.\boldsymbol{s} = \boldsymbol{q}_y$, respectively. Then, the pointwise maximum $\boldsymbol{q}_m = \max(\boldsymbol{q}_x, \boldsymbol{q}_y)$ is also reachable from both states $\boldsymbol{q}_x$ and $\boldsymbol{q}_y$ via the two rules r_y^1 and r_x^1 given in Definition 4.7, i.e., $\boldsymbol{q}_x \xrightarrow{r_y^1} \boldsymbol{q}_m$ and $\boldsymbol{q}_y \xrightarrow{r_x^1} \boldsymbol{q}_m$.*

Proof: Without loss of generality r_x is selected to be executed first. Therefore, the cluster is in state $\boldsymbol{q}_x$ and r_y^1 is selected to continue execution.[11] For r_y^1 to execute its precondition $r_y^1.\boldsymbol{l} \leq \boldsymbol{q}_x \leq r_y^1.\boldsymbol{u}$ has to be satisfied. First the lower bound $r_y^1.\boldsymbol{l} \leq \boldsymbol{q}_x$ is considered. With (Definition 4.7) $r_y^1.\boldsymbol{l} = r_y.\boldsymbol{l} + \min(r_y.\boldsymbol{s}, r_x.\boldsymbol{s}) \leq r_y.\boldsymbol{l} + r_x.\boldsymbol{s} \leq \boldsymbol{q} + r_x.\boldsymbol{s} = \boldsymbol{q}_x$ the lower bound is handled. For the upper bound $\boldsymbol{q}_x \leq r_y^1.\boldsymbol{u}$, two cases can occur for an actor a: First, $r_y.s_a \leq r_x.s_a$. In this case, $r_y^1.s_a = r_y.s_a - \min(r_y.s_a, r_x.s_a) = 0$, and, by construction $q_{x,a} \leq r_y^1.u_a = \infty$. Otherwise, $r_y.s_a > r_x.s_a$, and, by construction $r_y^1.u_a = r_y.u_a + \min(r_y.s_a, r_x.s_a) = r_y.u_a + r_x.s_a \geq q_a + r_x.s_a = q_{x,a}$. After proving the precondition, r_y^1 is executed and the resulting state $\boldsymbol{q}'$ is calculated to be $\boldsymbol{q}' = \boldsymbol{q}_x + r_y.\boldsymbol{s} - \min(r_y.\boldsymbol{s}, r_x.\boldsymbol{s}) = \boldsymbol{q} + r_x.\boldsymbol{s} + r_y.\boldsymbol{s} - \min(r_y.\boldsymbol{s}, r_x.\boldsymbol{s}) = \boldsymbol{q} + \max(r_y.\boldsymbol{s}, r_x.\boldsymbol{s}) = \max(\boldsymbol{q} + r_y.\boldsymbol{s}, \boldsymbol{q} + r_x.\boldsymbol{s}) = \max(\boldsymbol{q}_y, \boldsymbol{q}_x) = \boldsymbol{q}_m$. □

In the following, the general case with arbitrary $\boldsymbol{q}_x, \boldsymbol{q}_y \in Q$ is discussed. The lemma only requires that there is a common state $\boldsymbol{q}_0$ from which both $\boldsymbol{q}_x$ and $\boldsymbol{q}_y$ are rule reachable. Note that this is trivially true for the *input/output dependency states* $\boldsymbol{q} \in Q_{io}$ as the derivation of initial rules in Section 4.2 ensures that for each $\boldsymbol{q} \in Q_{io}$ a sequence from $\boldsymbol{q}_\perp$ exists. An equivalent argument to the one given at the end of Section 5.1 for the extension of the reachability of all states $q \in Q$ from the reachability of the *input/output dependency states* $\boldsymbol{q} \in Q_{io}$ can be applied to extend the rule reachability from the *input/output dependency states* to all states. However, this argument requires a rule equivalent for Lemma 5.1 which is given below:

Lemma 5.4 (Multi-Step Max State Rule Reachability). *For every two states $\boldsymbol{q}_x, \boldsymbol{q}_y \in Q$ which are rule reachable from a common state $\boldsymbol{q}_0$, i.e., $\boldsymbol{q}_0 \xrightarrow{s_x} \boldsymbol{q}_x$ and $\boldsymbol{q}_0 \xrightarrow{s_y} \boldsymbol{q}_y$, the state $\boldsymbol{q}_m = \max(\boldsymbol{q}_x, \boldsymbol{q}_y)$ is rule reachable from both $\boldsymbol{q}_x$ and $\boldsymbol{q}_y$ via sequences $s_{x,m}$ and $s_{y,m}$, i.e., $\boldsymbol{q}_x \xrightarrow{s_{x,m}} \boldsymbol{q}_m$ and $\boldsymbol{q}_y \xrightarrow{s_{y,m}} \boldsymbol{q}_m$.*

[11] The existence of a valid schedule $\nu_{x,m}$ for r_y^1 to execute has been proven by Lemma 5.1.

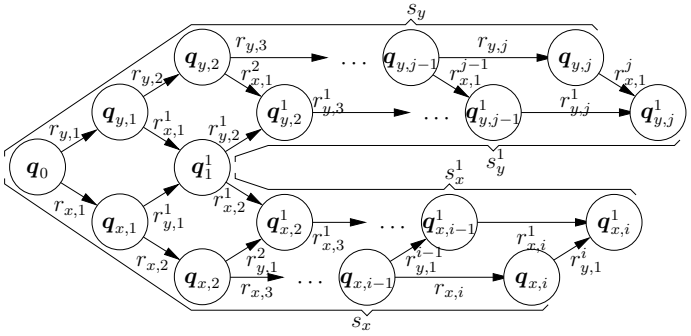


Fig. 9. Rule application sequences used in the proof of the *Multi-Step Max State Rule Reachability*.

Proof: The lemma is proven via induction. Given that $\boldsymbol{q}_x = \boldsymbol{q}_{x,i}$ and $\boldsymbol{q}_y = \boldsymbol{q}_{y,j}$ (cf. Fig. 9) are reachable from a state $\boldsymbol{q}_0 = \boldsymbol{q}_{x,0} = \boldsymbol{q}_{y,0}$ via two sequences s_x ($\boldsymbol{q}_{x,0} \xrightarrow{r_{x,1}} \boldsymbol{q}_{x,1} \xrightarrow{r_{x,2}} \ldots \xrightarrow{r_{x,i}} \boldsymbol{q}_{x,i}$) and s_y ($\boldsymbol{q}_{y,0} \xrightarrow{r_{y,1}} \boldsymbol{q}_{y,1} \xrightarrow{r_{y,2}} \ldots \xrightarrow{r_{y,j}} \boldsymbol{q}_{y,j}$). The maximum indices i and j denote the length, i.e., the number of rule applications, for the sequences s_x and s_y, respectively.

First note that for the case of i being zero the lemma is trivially true, i.e., if i is zero then $\boldsymbol{q}_x = \boldsymbol{q}_0$ and $\boldsymbol{q}_y \geq \boldsymbol{q}_x$, therefore $\boldsymbol{q}_m = \boldsymbol{q}_y$ which is reachable from $\boldsymbol{q}_0 = \boldsymbol{q}_x$ via the sequence s_y. For j being zero the symmetrical argument applies.

Hence, it can be assumed that $i, j \geq 1$ and that the lemma is true for sequences s^1_x ($\boldsymbol{q}^1_1 = \boldsymbol{q}^1_{x,1} \xrightarrow{r^1_{x,2}} \boldsymbol{q}^1_{x,2} \xrightarrow{r^1_{x,3}} \ldots \xrightarrow{r^1_{x,i}} \boldsymbol{q}^1_{x,i} = \boldsymbol{q}^1_x$) of length $i-1$ and s^1_y ($\boldsymbol{q}^1_1 = \boldsymbol{q}^1_{y,1} \xrightarrow{r^1_{y,2}} \boldsymbol{q}^1_{y,2} \xrightarrow{r^1_{y,3}} \ldots \xrightarrow{r^1_{y,j}} \boldsymbol{q}^1_{y,j} = \boldsymbol{q}^1_y$) of length $j-1$. The sequences s^1_x and s^1_y are used as induction hypothesis for the proof.

The induction proceeds by proving the existence of two such sequences s^1_x and s^1_y, such that $\boldsymbol{q}_m = \max(\boldsymbol{q}^1_x, \boldsymbol{q}^1_y)$ and the existence of two rules $\boldsymbol{q}_x \xrightarrow{r^i_{y,1}} \boldsymbol{q}^1_x$ and $\boldsymbol{q}_y \xrightarrow{r^j_{x,1}} \boldsymbol{q}^1_y$. With a successful proof of the existence of $s^1_x, r^i_{y,1}$ and $s^1_y, r^j_{x,1}$ the induction hypothesis can be applied and the proof of Lemma 5.4 is finished.

First, $\max(\boldsymbol{q}_{x,1}, \boldsymbol{q}_{y,1})$ is chosen for $\boldsymbol{q}^1_1$. Without loss of generality only the existence of s^1_x is proven. Note that Lemma 5.3 applies to the rules $r_{x,1}$ and $r_{y,1}$, therefore the presence of the rule $r^1_{y,1}$ is ensured. Now, assuming the presence of the rule $\boldsymbol{q}_{x,n-1} \xrightarrow{r^{n-1}_{y,1}} \boldsymbol{q}^1_{x,n-2}$ note that Lemma 5.3 applies to the rules $r^{n-1}_{y,1}$ and $r_{x,n}$ which results in the rules $r^n_{y,1}$ and $r^1_{x,n}$. Furthermore, $\boldsymbol{q}^1_x = \max(\boldsymbol{q}_{y,1}, \boldsymbol{q}_{x,1}, \boldsymbol{q}_{x,2}, \ldots \boldsymbol{q}_{x,i}) = \max(\boldsymbol{q}_{y,1}, \boldsymbol{q}_{x,i}) = \max(\boldsymbol{q}_{y,1}, \boldsymbol{q}_x)$. The symmetrical argument applies for s^1_y with $\boldsymbol{q}^1_y = \max(\boldsymbol{q}_{x,1}, \boldsymbol{q}_y)$. Finally, note that $\max(\boldsymbol{q}^1_x, \boldsymbol{q}^1_y) = \max(\boldsymbol{q}_{y,1}, \boldsymbol{q}_x, \boldsymbol{q}_{x,1}, \boldsymbol{q}_y) = \max(\boldsymbol{q}_x, \boldsymbol{q}_y) = \boldsymbol{q}_m$. □

5.3. Semantic Equivalence

In this section, the correctness of the presented clustering algorithm is finally proven by showing the equivalence of the semantics of a static dataflow subgraph g_S and its corresponding composite actor a_c. This is done by defining the semantics of both, g_S and a_c, with the denotational semantics for *Kahn process network (KPN)* [Kahn 1974] and abstracting from data values. The actual proof is done by induction.

THEOREM 5.5. *Given a static dataflow subgraph g_S of a dataflow graph g satisfying the clustering condition in Definition 4.10 and its corresponding composite actor a_c constructed*

by the clustering algorithm in Section 4. It holds that the behaviors of g and the dataflow graph $\tilde{g}$ resulting from replacing g_S with a_c are sequence equivalent.[12]

Proof: Theorem 5.5 is proven by showing that the Kahn descriptions F_{g_S} and F_{a_c} of g_S and a_c are equivalent, i.e., $F_{g_\mathrm{S}} \equiv F_{a_\mathrm{c}}$.[13] These functions map tuples of sequences of tokens on the input ports to tuples of sequences of tokens on the output ports.

However, due to the data independent nature of SDF and CSDF actors, the sequences can be abstracted to their length. The production of the correct token values is guaranteed by the firing of the contained actors on the same tokens as in the original subgraph. Finally, the sequence length produced by an input or output actor can be represented by the number of input or output actor firings. With these abstractions, the functions $F'_{g_\mathrm{S}}, F'_{a_\mathrm{c}} : {\mathbb{N}_0}^{|A_I|} \rightarrow {\mathbb{N}_0}^{|A_O|}$ are used, which map the number of input actor firings into the number of output actor firings. Thus, the equivalence of the denotational Kahn functions is reduced to the equivalence of their corresponding abstracted functions, i.e., $F_{g_\mathrm{S}} \equiv F_{a_\mathrm{c}} \iff F'_{g_\mathrm{S}} \equiv F'_{a_\mathrm{c}}$.

Note that for all inputs F'_{a_c} can only be less than F'_{g_S} as Lemma 5.1 ensures that all rules execute valid schedules, i.e., the non-equivalence of F'_{g_S} and F'_{a_c} could only be caused by missing output actor firings of F'_{a_c}. The proof proceeds by showing a contradiction. Assume that for some $\boldsymbol{q}_\mathrm{in} \in {\mathbb{N}_0}^{|A_\mathrm{I}|}$ the composite actor lacks output actor firings, i.e., $F'_{a_\mathrm{c}}(\boldsymbol{q}_\mathrm{in}) < F'_{g_\mathrm{S}}(\boldsymbol{q}_\mathrm{in})$, and that no sequence s exists adding these missing output actor firings. Let the state $\boldsymbol{q}_\mathrm{def} = (\boldsymbol{q}_\mathrm{in}, F'_{a_\mathrm{c}}(\boldsymbol{q}_\mathrm{in}))$ be this defective state and $\boldsymbol{q}_\mathrm{ok} = (\boldsymbol{q}_\mathrm{in}, F'_{g_\mathrm{S}}(\boldsymbol{q}_\mathrm{in}))$ the correct state. Then there exists at least one output actor a_m with missing firings, i.e., $q_{\mathrm{def},a_m} < q_{\mathrm{ok},a_m}$, and a corresponding initial input/output dependency state $\boldsymbol{q}_\mathrm{io} \in Q_\mathrm{io}$ where this output is maximized, i.e., $q_{\mathrm{io},a_m} = q_{\mathrm{ok},a_m}$. Then, by the least fixpoint operation from Definition 4.4 the state $\boldsymbol{q}_\mathrm{corr} = \max(\boldsymbol{q}_\mathrm{io}, \boldsymbol{q}_\mathrm{def})$ must also exist. Furthermore, by Lemma 5.4 there exists a sequence $\boldsymbol{q}_\mathrm{def} \xrightarrow{s} \boldsymbol{q}_\mathrm{corr}$. If there are still output actor firings missing (from output actors other than a_m) the previous steps can be repeated until $\boldsymbol{q}_\mathrm{ok}$ is reached. □

6. RESULTS

In this section, binary executables have been generated by (M1) fully dynamically scheduling the graphs (for reference purpose), (M2) the presented rule-based approach, and (M3) the FSM approach proposed in [Falk et al. 2008]. These binary executables are used for speedup and code size measurements to evaluate the approaches against each other.

First the dynamic scheduling algorithm is presented which is used as a baseline for all comparisons. This scheduling algorithm is a variant of the simple round-robin scheduler with the modification that (a) each actor is fired in a loop until the number of tokens on its inputs are insufficient for further firings and (b) all static actors in a cluster g_S are scheduled by a round-robin subscheduler which is executed by the main round-robin scheduler (cf. lines 14-22 from Algorithm 1).

Modification (b) enabled the replacement of the round-robin subscheduler with the FSM approach or the rule-based approach without any modifications to the check and execute sequence of the dynamic actors by the main round-robin scheduler. This is important as permutations in the check and execute sequence can lead to noticeable performance fluctuations unrelated to the quasi-static scheduling under consideration. The replacement subscheduler to execute the rules derived in Section 4 is given below in Algorithm 2. This subscheduler is used to replace lines 14-22 from Algorithm 1. The resulting scheduler is used to evaluate the performance and code size of the rule-based static dataflow clustering algorithm (M2) presented in this article. For evaluation of the FSM approach (M3) lines 14-22 from Algo-

[12]The definition of sequence equivalence can be found in [Lee and Sangiovanni-Vincentelli 1998].

[13]The relation between Kahn's denotational semantics and the semantics of dataflow models with the notion of firing is presented in [Lee 1997].

ALGORITHM 1: Fully dynamic scheduler

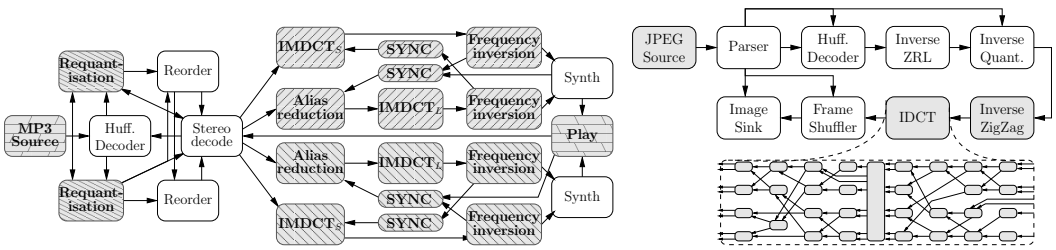
```
1  VAR: fired_D, fired_S ∈ {true, false}
2  VAR: q_c ∈ Q Current cluster state used by the rule subscheduler
3  IN:  The set of actors A and the static subset A_S ⊆ A
4  BEGIN
5     LET q_c ← q_⊥
6     DO
7        LET fired_D ← false
8        FOREACH a_D ∈ A − A_S DO
9           IF can fire a_D THEN
10             LET fired_D ← true
11             DO fire a_D WHILE can fire a_D
12          ENDIF
13       ENDFOR
14       DO
15          LET fired_S ← false
16          FOREACH a_S ∈ actorsing_S DO
17             IF can fire a_S THEN
18                LET fired_S ← true
19                DO fire a_S WHILE can fire a_S
20             ENDIF
21          ENDFOR
22       WHILE fired_S
23       LET fired_D ← fired_S ∨ fired_D
24    WHILE fired_D
25 END
```

rithm 1 are replaced with the appropriate subscheduler to execute the cluster FSM derived via the algorithm given in [Falk et al. 2008].

ALGORITHM 2: Rule subscheduler for clustered actors

```
13       ...
14       DO
15          LET fired_S ← false
16          FOREACH r ∈ R DO
17             IF r.l ≤ q_c ≤ r.u ∧ enough inputs for r.s THEN
18                LET fired_S ← true
19                LET q_c ← q_c + r.s
20                fire static schedule for r.s
21             ENDIF
22          ENDFOR
23          IF q_c ≥ μ_io THEN
24             LET q_c ← q_c − μ_io
25          ENDIF
26       WHILE fired_S
27       ...
```

In order to illustrate the benefits of the clustering algorithm developed in Section 4, it has been applied to both real world applications as well as synthetic dataflow graphs. The first real world application is an mp3 decoder working on the *mp3 granule level* (i.e., 576 frequency/time domain samples) as shown in Fig. 10a). For two subgraphs QSSs have been



a) mp3 decoder b) Motion-JPEG decoder

Fig. 10. Dataflow graph of two real world applications. Shaded vertices correspond to static actors. White vertices are dynamic actors. a) The mp3 decoder has five static subgraphs (two subgraphs consist of a single actor only and one subgraph only containing two actors). The remaining two subgraphs are processed by the presented clustering algorithm to compute their QSS schedules. b) The Motion-JPEG decoder has two static subgraphs (the source actor being the trivial subgraph). As can be seen the IDCT and the inverse ZigZag transformation correspond to the largest static subgraph. Hence, the presented clustering methodology has been applied to this subgraph.

computed using our proposed approach. To evaluate the scheduling overhead reduction in isolation, we first removed as much functionality as possible from all actors. The dynamic scheduler for this mp3 decoder required about 184 ms runtime for a given mp3 input stream (approx. 20 MB). This decreases to 57 ms when using the previously computed QSSs for the two subgraphs, i.e., an improvement of approx. 69%. For a real world test we switched back to the unmodified mp3 decoder. This resulted in about 2,230 ms decoding time for the same input stream using the dynamic scheduler, and 2,100 ms when using the QS schedules. This corresponds to an improvement of approx. 6%. However, for dataflow graphs with more fine grained actors, the achievable scheduling overhead reduction is expected to be higher.

The Motion-JPEG decoder depicted in Fig. 10b) where each butterfly operation of the 8×8 inverse discrete cosine transform ($IDCT$) is modeled as an SDF actor. Such fine-grained graphs can result from the usage of tools like [Venkataramani et al. 2004] which replace coarse-grained actors in a data-flow graph by fine-grained subgraphs corresponding to data flow graphs derived from the functions of the actors. This expanded graph can then be clustered again to better utilize the resources of an MPSoC environment than would be feasible by only clustering actors atomically.

For test purposes the standard Lena image was used and the decoding was repeated 25 times. For this test case an improvement of 89% could be observed where the dynamic scheduler (M1) required 329 ms whereas after applying the presented rule-based QSS scheduling (M2) only 35 ms where needed. With the functionality restored, an improvement of 40% could still be observed where the dynamic scheduler (M1) had a runtime of 823 ms while only 492 ms where required by the rule-based QSS (M2).

In order to evaluate the presented clustering algorithm more thoroughly, it was applied to 48000 randomly generated dataflow graphs: The generated graphs have the same properties, except (a) the average input and output degree of the dataflow graph vertices and (b) the number of actor firings necessary to reach the initial state of the SDF graph, i.e., the sum of the repetition vector scalars. The average input and output degree was varied from 2 to 7 in increments of 1 and the repetition vector sum from 100 to 1000 in increments of 300. Using the SDF3 tool [Stuijk et al. 2006], six different SDF graphs for each pair of average input/output degree and repetition vector sum were generated, thus creating 120 initial dataflow graphs. The generated graphs consist of 60 actors and are cyclic, i.e., they contain strongly connected components, with random but consistent SDF rates and sufficient initial tokens in order to guarantee a deadlock-free self-scheduled execution. For each random graph various *clusterings* were generated. A clustering is a set of clusters $\mathbf{C}$ into which the static actors A_S of the graph have been partitioned in such a way that each cluster

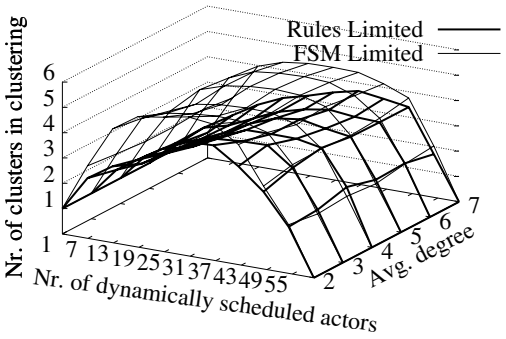


Fig. 11. Example of the average clusters in the fastest clustering under binary code size constraints for random graphs with a repetition vector sum of 100.

conforms to the cluster condition given in Definition 4.10. To execute such clusterings the dynamic scheduler from Algorithm 1 is simply equipped with the corresponding number of subschedulers (one for each cluster in the clustering). Furthermore, we can associate with each clustering its *number of dynamically scheduled actors* $sched_{\mathrm{dyn}}$.

$$sched_{\mathrm{dyn}} = |A - A_{\mathrm{S}}| + |\mathbf{C}|$$

As one can see from the definition above a cluster results in a composite actor which must be dynamically scheduled (the $|\mathbf{C}|$ part). In fact this exactly corresponds to the subschedulers in the dynamic scheduler and is also the number which is depicted on the x-axis of the speedup comparisons in Figure 12.

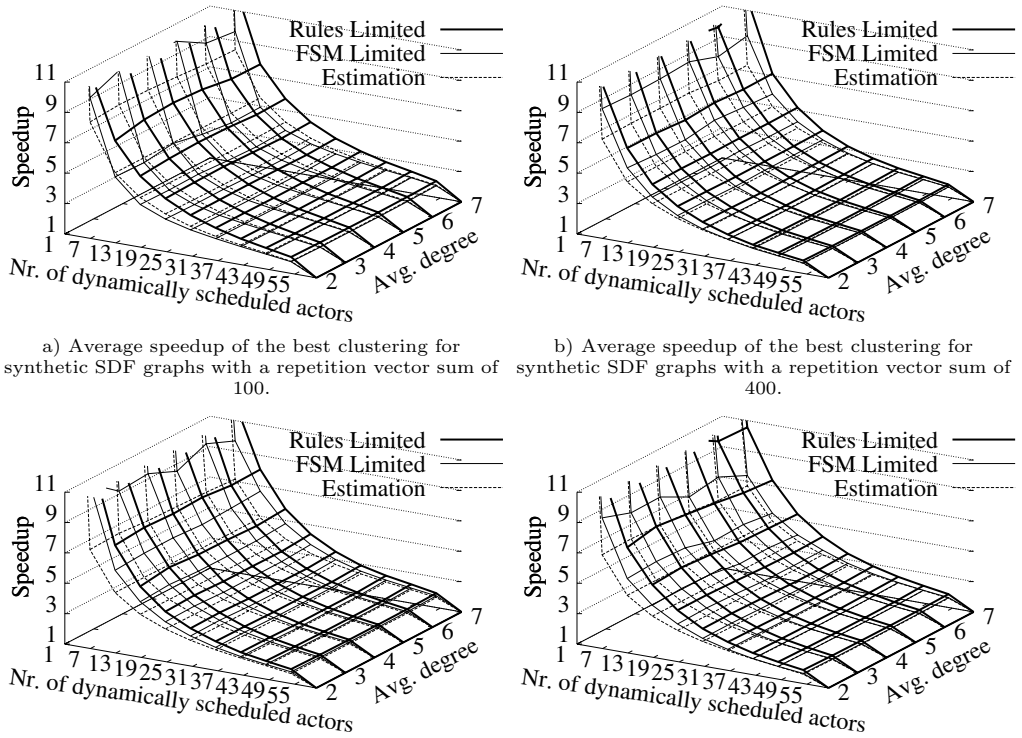
Frequently the best clustering of a static region A_{S} is not a monolithic composite actor a_{c}, if such a cluster is feasible at all due to the clustering condition, but a partitioning of the static region into a set of smaller composite actors all having less complex input/output behavior. An example of such cases can be seen in Figure 11. Also note that the binary code size constraints of the FSM approach (M3) forces the usage of more clusters in the clustering to satisfy the constraint of at most 133% increase in binary code size as compared to the fully dynamic scheduled system. This often yields a better speedup compared to a big composite actor.

Indeed, the speedup value (on the y-axis) for a given average input/output degree d (z-axis), number of dynamically scheduled actors $sched_{\mathrm{dyn}}$ (x-axis) and repetition vector sum rvs (Figure 12 a), b), c) or d)) is the average of the speedup values achieved by the best clustering with the given number of dynamically scheduled actors for each of the six different initial SDF graphs generated with the SDF3 tool which correspond to the given input/output degree d and repetition vector sum rvs.

To evaluate the improvement potential which can be exploited by the presented rules-based clustering approach, the scheduling overhead of the round-robin scheduler from Algorithm 1 was measured. Each initial dataflow graph has per definition no dynamic actors. Hence, each initial dataflow graph can be fully statically scheduled. The overhead s_{overhead} is defined as the factor by which the fully statically scheduled initial dataflow graph is faster than the same graph scheduled fully dynamically. As clustering reduces this scheduling overhead this represents an upper bound for the speedup that can be obtained by clustering.

However, if the clustering is between fully static and fully dynamic the remaining scheduling overhead in the system can only be estimated. The estimation assumes that the scheduling overhead is uniformly spread over all actors.

$$s_{\mathrm{estimation}} = \frac{s_{\mathrm{overhead}} \cdot 59}{s_{\mathrm{overhead}} \cdot (sched_{\mathrm{dyn}} - 1) + 60 - sched_{\mathrm{dyn}}}$$



a) Average speedup of the best clustering for synthetic SDF graphs with a repetition vector sum of 100.

b) Average speedup of the best clustering for synthetic SDF graphs with a repetition vector sum of 400.

c) Average speedup of the best clustering for synthetic SDF graphs with a repetition vector sum of 700.

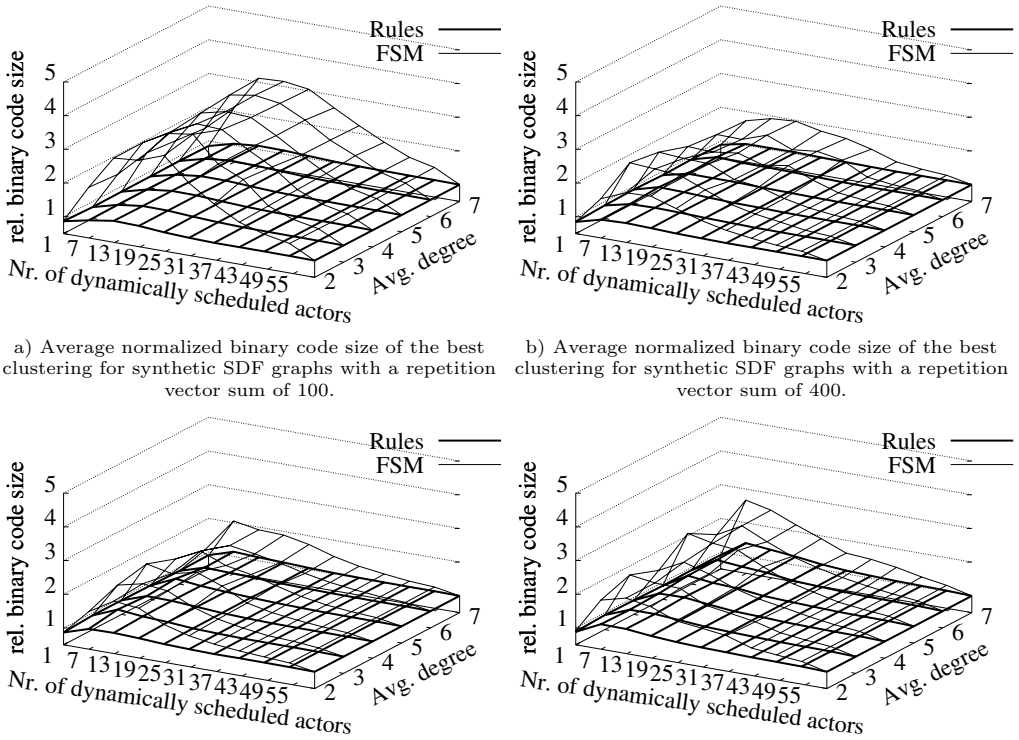
d) Average speedup of the best clustering for synthetic SDF graphs with a repetition vector sum of 1000.

Fig. 12. Average speedup of the best clustering for each of the six random test graphs with the given average input/output degree, repetition vector sum, and number of dynamically scheduled actors in the clustering. The measurements where conducted on an Intel® Core^{TM} i7-2600 CPU with 3.40GHz. All executables where generated with the `-Os` compile option of gcc and then stripped. Furthermore, the binary code sizes produced by the synthesizing the clustering with the rule-based approach (M2) and the FSM approach (M3) were also required to not exceed 133% of the code size of the dynamic case (M1).

This is only a coarse approximation as the actors have different repetition counts. As can be seen in Figure 12 for higher repetition vector sums the approximation gets worse. This is expected as the speedup of the rule-based approach (M2) and the FSM approach (M3) is the average of the best clustering for each random test graph. And the best clustering tends to select the actors for clustering which eliminates the most scheduling overhead from the system and not the average case assumed by the uniform spread of the scheduling overhead over all actors.

In embedded systems the available amount of memory is typically constrained. Hence, clusterings which when synthesized by the rule-based approach (M2) or the FSM approach (M3) exceed a binary code of of 133% of the code size of the dynamic case (M1) where excluded from consideration for the best clustering for the speedups as depicted in Figure 12. In case no such exclusion is enforced the relative binary code size of the rule-based approach (M2) or the FSM approach (M3) is depicted in Figure 13.

All executables where generated with the `-Os` compile option of gcc and then stripped. After that, the remaining executable sizes where measured. The executable size for the fully dynamically scheduled system (M1) is the reference and corresponds to 1. One can see a correlation between Figure 13 and Figure 12. In cases where the rel. binary code size of the FSM approach (M3) far exceeds the binary code size of the dynamically scheduled system



a) Average normalized binary code size of the best clustering for synthetic SDF graphs with a repetition vector sum of 100.

b) Average normalized binary code size of the best clustering for synthetic SDF graphs with a repetition vector sum of 400.

c) Average normalized binary code size of the best clustering for synthetic SDF graphs with a repetition vector sum of 700.

d) Average normalized binary code size of the best clustering for synthetic SDF graphs with a repetition vector sum of 1000.

Fig. 13. Average normalized binary code size of the best clustering for each of the six random test graphs with the given average input/output degree, repetition vector sum, and number of dynamically scheduled actors in the clustering. No code size limitations where applied to find the clusterings withe the best speedup.

(M1) the speedup of the FSM approach (M3) under code size limitations also falls noticeably behind the speedup of the rule based approach (M2). This blowup stems from the fact that the representation of QSSs via FSMs requires huge code sizes for complex schedules.

7. CONCLUSIONS

In this article, a generalized clustering approach for static dataflow subgraphs has been presented. The clustering algorithm computes a quasi-static schedule reducing the scheduling overhead for one processor of an MPSoC while still accommodating a worst-case environment of the cluster. Through the proposed clustering approach, the scheduling of these subgraphs can be coordinated with enclosing system representations in a way that systematically exploits the predictability and efficiency of the static dataflow model. This greatly enhances the power of our techniques in terms of avoiding deadlock, increasing the design space for clustering, and providing for integration with more general models of computation. We have shown benefits of up to 40% throughput improvement for real world examples. Future work will focus on optimal clustering techniques, i.e., to identify 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

Alur, R., Brayton, R. K., Henzinger, T. A., Qadeer, S., and Rajamani, S. K. 2001. Partial-order reduction in symbolic state-space exploration. *Form. Methods Syst. Des. 18,* 2, 97–116.

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, 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. 1988. Software synthesis and code generation for signal processing systems. *Philosophy of Science 55*, 614–630.

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*.

Bilsen, G., Engels, M., Lauwereins, R., and Peperstraete, J. 1996. Cyclo-Static Dataflow. *IEEE Transaction on Signal Processing 44,* 2, 397–408.

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.*

Eker, J. and Janneck, J. 2003. Cal language report: Specification of the cal actor language. Tech. rep., Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California.

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).

Falk, J., Zebelein, C., Haubelt, C., and Teich, J. 2011. A Rule-Based Static Dataflow Clustering Algorithm for Efficient Embedded Software Synthesis. In *In Proceedings of Design, Automation and Test in Europe (DATE'11)* (March 14–18).

Gerstlauer, A., Haubelt, C., Pimentel, A., Stefanov, T., Gajski, D., and Teich, J. 2009. Electronic system-level synthesis methodologies. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 28,* 10, 1517–1530.

Gu, R., Janneck, J. W., Raulet, M., and Bhattacharyya, S. S. 2011. Exploiting statically schedulable regions in dataflow programs. *Journal of Signal Processing Systems 63,* 1, 129–142.

Ha, S., Kim, S., Lee, C., Yi, Y., Kwon, S., and Joo, Y.-P. 2007. PeaCE: A Hardware-Software Codesign Environment of Multimedia Embedded Systems. *ACM Trans. Design Automation of Electronic Systems 12,* 3, 1–25.

Haid, W., Huang, K., Bacivarov, I., and Thiele, L. 2009. Multiprocessor SoC software design flows. *Signal Processing Magazine, IEEE 26,* 6, 64 –71.

Haubelt, C., Falk, J., Keinert, J., Schlichter, T., Streubühr, M., Deyhle, A., Hadert, A., and Teich, J. 2007. A SystemC-based Design Methodology for Digital Signal Processing Systems. *EURASIP Journal on Embedded Systems, Special Issue on Embedded Digital Signal Processing Systems 2007*, Article ID 47580.

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.

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

This work has been partially supported by the German Science Foundation (DFG) under HA 4463/3-1.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

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.

Kwon, S., Kim, Y., Jeun, W.-C., Ha, S., and Paek, Y. 2008. A Retargetable Parallel Programming Framework for MPSoC. *ACM Trans. Design Automation of Electronic Systems 13,* 3.

Lee, E. A. 1997. A denotational semantics for dataflow with firing. Tech. rep., EECS, University of California, Berkeley, CA, USA 94720.

Lee, E. A. 2006. The problem with threads. *Computer 39*, 33–42.

Lee, E. A. and Messerschmitt, D. G. 1987b. Synchronous Data Flow. *Proceedings of the IEEE 75,* 9, 1235–1245.

Lee, E. A. and Messerschmitt, D. G. 1987a. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers C-36,* 1, 24–35.

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, 1217–1229.

Meijer, S., Nikolov, H., and Stefanov, T. 2010. Throughput modeling to evaluate process merging transformations in polyhedral process networks. In *DATE*. IEEE, 747–752.

Murata, T. 1989. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE 77,* 4, 541 –580.

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.

Plishker, W., Sane, N., and Bhattacharyya, S. 2009a. A generalized scheduling approach for dynamic dataflow applications. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.* 111–116.

Plishker, W., Sane, N., and Bhattacharyya, S. 2009b. Mode grouping for more effective generalized scheduling of dynamic dataflow applications. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*. 923–926.

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.*

Stuijk, S., Geilen, M., and Basten, T. 2006. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*. 276–278.

Thiele, L., Bacivarov, I., Haid, W., and Huang, K. 2007. Mapping applications to tiled multiprocessor embedded systems. In *ACSD '07: Proceedings of the Seventh International Conference on Application of Concurrency to System Design*. 29–40.

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.

Venkataramani, G., Budiu, M., Chelcea, T., and Goldstein, S. C. 2004. C to asynchronous dataflow circuits: An end-to-end toolflow. In *IEEE 13th International Workshop on Logic Synthesis (IWLS)*. Temecula, CA.