Software synthesis from dataflow schedule graphs

Kyunghun Lee¹ · Yaesop Lee¹ · Abhay Raina¹ · Yanzhou Liu¹ · Jiahao Wu¹ · Christopher Defrancisci¹ · Benjamin S. Riggan² · Shuvra S. Bhattacharyya¹

Received: 10 April 2020 / Accepted: 30 December 2020 / Published online: 16 January 2021
© The Author(s) 2021

Abstract

The dataflow-model of computation is widely used in design and implementation of signal processing systems. In dataflow-based design processes, scheduling—the assignment and coordination of computational modules across processing resources—is a critical task that affects practical measures of performance, including latency, throughput, energy consumption, and memory requirements. Dataflow schedule graphs (DSGs) provide a formal abstraction for representing schedules in dataflow-based design processes. The DSG abstraction allows designers to model a schedule as a separate dataflow graph, thereby providing a formal, abstract (platform- and language-independent) representation for the schedule. In this paper, we introduce a design methodology that is based on explicit specifications of application graphs and schedules as cooperating dataflow models. We also develop new techniques and tools for automatically synthesizing efficient implementations on multicore platforms from these coupled application and schedule models. We demonstrate the proposed methodology and synthesis techniques through a case study involving real-time detection of people and vehicles using acoustic and seismic sensors.

Keywords Dataflow · Scheduling · Signal processing systems · Software synthesis

1 Introduction

In the design and implementation of digital signal processing (DSP) systems, dataflow is recognized as a natural model for specifying applications, and dataflow enables useful model-based methodologies for analysis, synthesis, and optimization of implementations. A wide range of embedded signal and information processing applications can be designed efficiently using the high level abstractions that are provided by dataflow programming models (e.g., see [1]).

In dataflow-based modeling of DSP systems, applications are represented as directed graphs in which vertices correspond to signal processing hardware/software modules, such as digital filters and fast Fourier transforms (FFTs), and each edge represents the flow of data from the output of one vertex to the input of another. Vertices in

DSP-oriented dataflow graphs are referred to as *actors*. Edges in dataflow graphs can be viewed as first in, first out (FIFO) channels that buffer data as it passes between pairs of communicating actors.

Dataflow representations are useful in exposing parallelism in programs, which has motivated their extensive study in the context of parallel computation (e.g., see [16]). In addition to their use in parallelizing computations for faster execution, dataflow graphs have additional advantages that stem from their modularity and formal foundation. For example, dataflow models can be applied to optimize memory requirements, enhance portability, and improve energy efficiency [1].

In addition to the directed graph structure of dataflow representations, another distinctive feature in dataflow is that actor execution is decomposed naturally into discrete units, which are called *firings* of the associated

✉ Kyunghun Lee, leekh3@umd.edu | ¹University of Maryland, College Park, MD, USA. ²U.S. Army Research Laboratory, Adelphi, MD, USA.

actor [23]. Each firing of an actor A consumes a well defined amount of data values (referred to as *tokens*) from its input ports and produces a well defined number of tokens on its output ports. These numbers of tokens produced and consumed are referred to as the *production and consumption rates* that are associated with the given firings and actor ports. Information about production and consumption rates is often of great relevance in deriving efficient implementations from dataflow graphs [1].

An important problem in the development of dataflow-based design tools is the automated synthesis of software from dataflow representations. Many software synthesis environments for dataflow environments have been presented in the literature (e.g., see [13, 31, 33, 34, 42, 48]), and dataflow-based software synthesis continues to be an active area of research in the embedded systems, electronic design automation, and signal processing communities. In this paper, we develop new software synthesis techniques for dataflow-based design and implementation of signal processing systems.

An important task in software synthesis from dataflow graphs is that of *scheduling*. Scheduling refers to the assignment of actors to processing resources and the ordering of actors that share the same resource. Scheduling typically involves very complex design spaces, and has significant impact on most relevant implementation metrics, including latency, throughput, energy consumption, and memory requirements. Our work in this paper builds on a model-based representation, called the *dataflow schedule graph (DSG)*, which has been introduced in prior work to model schedules for dataflow graphs [47]. The DSG approach allows designers to model a schedule for a dataflow graph as a separate dataflow graph, thereby providing a formal, abstract (platform- and language-independent) representation for the schedule.

A distinguishing aspect of our approach to software synthesis, compared to most related software synthesis techniques, is that we leave the design of the schedule up to the designer rather than generating the schedule automatically as part of the synthesis process. The schedule design is modeled by the designer using the DSG model described above.

Requiring the designer to specify the schedule has the disadvantage of requiring more effort by the designer, but offers the advantage of providing flexibility to designers who wish to have more control over the implementation process or who wish to target architectures or design constraints that are not well supported by available, fully-automated software synthesis processes. Thus, while at first the concept of a designer-specified schedule may seem to be purely a limitation, it in general actually represents a trade-off. This paper represents an effort to investigate the side of

this trade-off that favors giving more control and flexibility to designers.

Given the undecidable or NP-hard nature of most dataflow scheduling problems (e.g., see [1]), it is useful to have a structured means for the designer to override the solutions provided by automated schedulers that are available in his or her toolset. This capability, provided by the methods developed in this paper, does not diminish the importance of automated schedulers. Instead, it provides a complementary approach that can be used when automated schedulers are not available for the specific constraints or objectives of interest or when the solutions delivered by automated schedulers are not sufficient. The methods developed in this paper can therefore be viewed as complementary to the large body of prior work that has been developed on automated scheduling techniques for dataflow graphs, such as those developed by Jones et al. [17], Tendulkar [46], Chen and Zhou [9], and Ma and Sakellariou [29] to list a representative collection of such prior work. More discussion about relationships to prior work in scheduling and software synthesis is discussed in Sect. 3.

More specifically, in this paper, we develop methods to synthesize embedded software for implementing schedules from abstract DSG representations of the schedules. While we demonstrate this software synthesis capability by translating DSGs into C language implementations, the use of a model-based schedule representation makes the approach readily retargetable to other implementation languages. We also investigate a number of optimization techniques to improve the efficiency of software synthesized from DSGs. We experiment with our proposed new software synthesis techniques by implementing them in the dataflow interchange format (DIF) environment, which is a software tool that enables experimentation with new kinds of dataflow-based techniques for modeling, analysis, and optimization [15]. We demonstrate the correctness and efficiency of our software synthesis techniques through experimental evaluation of the generated software from a complex case study involving real-time detection of people and vehicles using acoustic and seismic sensors.

Portions of this work have been reported in partial/preliminary form in the M.S. thesis by Raina [39], and the Ph.D. thesis by Lee [24].

For the reader's reference, a legend of the abbreviations used in the paper is provided in an "Appendix" at the end of the paper.

2 Background

In this section, we review dataflow models and methods that are applied in the contributions presented in this paper.

2.1 DSPCAD framework

We prototype the software synthesis techniques introduced in this paper using a software toolset called the DSPCAD Framework [27]. The DSPCAD Framework is composed of three software packages—LIDE, DIF, and DICE—which address different aspects of the design and implementation process for embedded signal processing systems. DSPCAD is an acronym that stands (in reverse order) for Computer-Aided Design for Digital Signal Processing systems. In this section, we give a brief overview of the three main packages that are encapsulated within the DSPCAD Framework.

The DSPCAD Lightweight Dataflow Environment (LIDE) [41] is a flexible, lightweight design environment that facilitates design and implementation of DSP applications using dataflow techniques. LIDE provides dataflow libraries of *graph elements* (actors and edges), and a compact set of application programming interfaces (APIs) that allows designers to easily develop and integrate their own dataflow-based graph elements. The APIs of LIDE are defined abstractly in terms of fundamental dataflow principles, and can be mapped into concrete implementations in arbitrary programming languages. LIDE includes implementations of the APIs in C, C++, CUDA, OpenCL, and Verilog. In this work, we use LIDE-C, which is a sub-package of LIDE that is based on C language implementations of the LIDE APIs.

The dataflow interchange format (DIF) is a software tool that facilitates experimentation with dataflow-based techniques for modeling, analysis, and optimization [15]. Whereas LIDE provides capabilities for implementing actors, DIF provides capabilities for specifying and manipulating abstract dataflow graphs, where the focus is on graph- and dataflow-related properties, and details of the actors (graph vertices) are abstracted. The abstractions of the actors can take the form of arbitrary attributes, such as execution time estimates or production and consumption rates for the input and output ports. The DIF Package contains an implementation of (parser for) the DIF Language, which is a textual language for describing abstract dataflow graphs and attribute characterizations for dataflow graph elements. The DIF Package also contains an extensible library of data structure and algorithm implementations for analyzing, transforming, and generating code from abstract dataflow models. Most of the code generation capabilities currently within DIF assume that LIDE is used as the actor implementation language, although there is no fundamental limitation that DIF must only be used with LIDE-based implementations.

The DSPCAD Integrative Command Line Environment (DICE) is package that facilitates cross-platform design and implementation of embedded software and

firmware [3]. DICE is designed to be used in the Bash command line environment. The package can be used independently of LIDE and DIF, but various features in DICE, such as its utilities for unit testing, have significant synergy when used together with LIDE and DIF. DICE can be used on different operating systems, including Ubuntu, MacOS, Windows, and Raspberry PI Raspbian.

The remainder of this paper discusses some aspects of components and tools in the DSPCAD Framework specifically as they relate to prototyping the novel software synthesis methods introduced in this paper. For a general overview of the DSPCAD Framework, we refer the reader to [27].

2.2 Synchronous dataflow

Synchronous dataflow (SDF) is a special case of dataflow where the production and consumption rates of the actors are constant [22]. Figure 1 shows an example of an SDF graph. The numbers above the edges represent the numbers of tokens that are consumed from and produced onto each edge when its sink and source actors fire, respectively. For example, Actor 4 cannot fire until 4 tokens are available on each of the input edges e_1 , e_2 , and e_3 .

A properly-constructed SDF graph can be run indefinitely (e.g., by encapsulating software for the graph within an infinite loop). Moreover, such indefinite or unbounded execution can be performed with finite memory requirements that can be predicted at compile time [22]. This capability for indefinite execution under bounded memory is of great utility in embedded signal processing.

Fig. 1 An example of an SDF graph

2.3 Core functional dataflow

Some applications involve dynamics in the rates at which data is produced and consumed from actors. SDF is not adequate for working with these *dynamic dataflow* applications. Various forms of dynamic dataflow have been introduced to address this limitation (e.g., see [1]). These dynamic dataflow models of computation provide more generality compared to SDF in expressing application behavior.

Core functional dataflow (CFDF) is a specific dynamic dataflow model of computation that we apply in this paper [35]. In CFDF, each actor A has an associated set of *modes* $\mu(A)$, which can be viewed as alternative templates or operational states for firing the actor. Each firing of A is based on a unique mode $m \in \mu(A)$. An actor can have any positive integer number of modes. Each mode is associated with constant production and consumption rates (“dataflow rates”) on the actor ports. However, the production and consumption rates can vary across different modes, and the mode of an actor can change from one firing to the next.

Each CFDF actor A has a *current mode* $c(A) \in \mu(A)$, which can be viewed as part of its internal state. When a CFDF actor is fired, it executes based on its current mode, and produces and consumes data to/from its ports based on the constant dataflow rates associated with this mode. As part of each firing, a CFDF actor also determines its next mode, which in turn determines how much input data and output buffer space (on the actor output edges) must be available to launch the next firing of the actor. When the actor is fired again, this next mode becomes the actor's current mode, and its production and consumption rates are governed by that mode.

CFDF is a highly expressive dataflow model. For example, Plishker et al. [36] show that CFDF is as expressive as Boolean dataflow (BDF), which is known to be a Turing-complete model [7].

For more detailed background on CFDF, we refer the reader to [35, 36].

2.4 Dataflow schedule graphs

As mentioned in Sect. 1, scheduling is an important aspect in the process of mapping dataflow models into efficient implementations. Some scheduling techniques are oriented toward simplicity of scheduler implementation or fast generation of schedules, and do not incorporate optimization of the constructed schedules. Other schedulers, which we refer to as *optimizing schedulers*, are designed to optimize relevant metrics in the targeted software implementations. These metrics include latency, throughput,

code size, buffer memory requirements, and energy consumption.

A great deal of heterogeneity in dataflow-based design flows is brought about by this large and growing variety of scheduling techniques, along with the diversity in their targeted metrics, and the diversity in the hardware platforms on which the derived schedules are to be executed. To help manage this heterogeneity, the concept of a *model-based representation for dataflow schedules* is useful. This concept has been investigated in prior work, such as that by Wu et al. [47], and more recently, by Zebelein [49]. While conventional use of dataflow graphs in DSP system modeling is for the modeling of application behavior, model-based scheduling representations provide for abstract modeling of schedules.

In this paper, we apply a specific form of model-based schedule representation called the *dataflow schedule graph (DSG)* [47]. DSGs are used to model schedules for CFDF-based application representations. Like CFDF graphs, DSGs are based on dataflow semantics. When a DSG G_s is used to model the schedule for a CFDF-based application representation G_a , we distinguish the two graphs by referring to G_a as the *application graph* that is associated with the *schedule graph* (DSG) G_s . DSGs apply to the highly expressive CFDF model of computation. Thus, they are significantly more flexible compared to other model-based, dataflow schedule representations, such as the synchronization graph representation [4] and decision state modeling approach [10], which are restricted to SDF application graphs.

A DSG is either a *sequential DSG (SDSG)* or a *concurrent DSG (CDSG)* [47]. A CDSG is composed of multiple SDSGs, where communication between different SDSGs is carried out through special DSG actors that are devoted to inter-processor or inter-thread communication. In Sect. 4, we discuss implementation issues that are relevant to both SDSGs and CDSGs, and in Sect. 5, we focus on implementation issues that are specific to CDSGs.

2.5 RAs and SCAs

DSGs are constructed using two different types of actors, which are called *reference actors (RAs)* and *schedule control actors (SCAs)*. Each RA r is associated with a specific application graph actor, which is denoted by $ref(r)$ and called the *referenced actor* of r . Intuitively, the RA r can be viewed as a wrapper around the *guarded firing* of $ref(r)$ in its enclosing application graph. Here, by a guarded firing of an application graph actor, we mean the execution of the actor *if it is enabled*—that is, if it has sufficient data on its input buffers and sufficient empty space on its output buffers to support the firing [35]. If the actor is not

enabled, then a guarded firing is effectively a no-operation (NOP).

In contrast, a firing that is not guarded will unconditionally execute the actor (without checking for data availability at run-time). Such unconditional execution of the actor can be useful if it is known through some form of static or dynamic analysis that $ref(r)$ (in the application graph) will be enabled every time r (in the DSG) is executed. In Sect. 4.1, we describe how the guard associated with an RA can optionally be bypassed to effectively achieve such an unconditional execution.

An RA has two associated sub-functions, called the *pre* and *post* functions of the RA. These sub-functions provide placeholders for optional pre-processing and post-processing that can be performed prior to and after, respectively, the guarded firing of $ref(r)$ that is associated with each firing of r .

Intuitively, SCAs are used to perform actions to control the order in which subsets of RAs are fired. SCAs provide an extensible set of constructs to control the flow of RA firings, together with the guarded firings of their encapsulated referenced actors. For example, the loop SCA can be used to execute individual RAs or chains of connected RAs repeatedly until some pre-specified or data-dependent termination condition is met [47]. A pair of related SCAs that provide different control functionality is the *if* SCA together with the *fi* SCA. These SCAs can be used to conditionally execute portions of a DSG. Another pair of related SCAs is the *snd* SCA and *rec* SCA, which are used in CDSGs to communicate data across different SDSGs—in particular, these SCAs are used, respectively, for sending and receiving data across SDSGs.

2.6 Global token population property

An important property of SDSGs is that they contain *at most one token* at any given time. We refer to this as the *global token population* property of SDSGs. The global token population property guarantees that at any given time during execution of a correctly-constructed SDSG, either (a) there are no tokens on any of the FIFOs in the SDSG or (b) there is exactly one nonempty FIFO, and this FIFO contains exactly one token. In general, case (a) may occur *during* the firing of a DSG actor—that is, after its input has been consumed and before any corresponding output has been produced.

The global token population property is ensured by construction—that is, by specific design rules that RAs, SCAs, and their connections must conform to.

In general, the global token population property applies only to SDSGs. In CDSGs, the property applies separately to each SDSG that is contained in the CDSG. However, it does not apply to edges in CDSGs whose source and sink

vertices reside in different SDSGs—for example, it does not apply to CDSG edges that are directed from a *snd* SCA to its corresponding *rec* SCA.

For more details on fundamental DSG concepts, including RAs and different types of SCAs, we refer the reader to [47].

3 Related work

Various prior research efforts are related to the problem of modeling of dataflow schedules. For example, Lee and Ha presented four classes of alternative scheduling strategies for real-time DSP systems [21]. These classes include, in decreasing order of flexibility, fully dynamic, static assignment, self-timed, and fully static scheduling. Ko et al. proposed a representation called the generalized schedule tree (GST) to represent a class of schedules called looped schedules [19]. The synchronization graph is a schedule representation for modeling self-timed, multiprocessor schedules for homogeneous SDF graphs [4]. The model facilitates optimization of inter-processor communication and synchronization for this class of schedules. The decision state modeling approach [10] provides a model-based schedule representation that is formulated directly on general SDF graphs, and does not require expansion to homogeneous SDF form. An extensive review of model-based schedule representations is provided in [2].

This paper is based on the dataflow schedule graph (DSG) representation introduced in [47]. The DSG went beyond previously-developed schedule models in its handling of dynamic scheduling and dynamic dataflow application behavior within a unified, dataflow-based schedule representation. DSGs are capable of representing a large class of static, dynamic, and quasi-static schedules. DSGs are also capable of representing both single and multiprocessor schedules.

This paper introduces novel software synthesis capabilities that help to automate the use of DSGs within design flows for model-based signal processing. In addition, this paper presents the first development of multicore implementation techniques using DSGs, and integrates these techniques into a new software synthesis tool, called DIF-DSG. DIF-DSG synthesizes software for single- and multi-threaded implementations from DSG representations. We also demonstrate (Sect. 7.3) a new method, called DSG token tracking (DTT), for executing DSGs efficiently. The DTT method is incorporated into the run-time library that accompanies DIF-DSG. All of these are novel contributions of this work, which build on the DSG modeling concepts introduced in [47].

Ragan-Kelley et al. present a domain specific language, called Halide, which supports decoupling of schedules

from algorithms for image processing pipelines. Halide applies a similar concept to that employed in the DSG approach; however, it operates at a lower level of abstraction [38]. In particular, Halide allows the programmer to specify schedules associated with individual functions, while the scheduling across functions is left to the compiler. In contrast, in the proposed DSG-based approach, the schedule for the overall application dataflow graph is provided by the designer, and optimization for individual actors is left up to the language-specific tools (e.g., C or C++ compilers) that are applied to the code produced by DIF-DSG software synthesis. In principle, Halide is complementary to the DSG approach in the sense that Halide can be used as the actor implementation language, while DSGs are used for graph-level schedule design. Such an integration is an interesting direction for future research.

Zebelein et al. present a unified model in which dataflow graphs and their schedules are represented together within the same model-based representation [49]. In this model, control flow associated with schedules is represented through composite actors that control execution of their encapsulated subsystems. This unified, hierarchical modeling approach is significantly different from the distinct application graph and schedule graph models that are used in the DSG model. This separation is useful to promote orthogonality in the design process. In particular, the DSG approach allows schedules and application graphs and their associated hierarchies to be designed, represented, and manipulated with strong separation of concerns. For general background on the utility of orthogonalization in system-level design, we refer the reader to [18]. Orthogonalization is an important consideration in the design of the DSPCAD Framework [27], which we reviewed in Sect. 2.1.

A large body of work exists on software synthesis for dataflow graphs. For example, Sung and Ha develop techniques for reducing the code size and buffer memory requirements in the PeaCE environment for software synthesis [45]. They also incorporate an approach to trade off some amount of code size compactness so as to minimize the total memory requirement for both code and data. Roquier et al. develop a framework for joint hardware and software synthesis starting from an extended version of the dataflow process network model of computation [40]. Oh and Ha develop software synthesis techniques for minimizing buffer memory requirements based on a decomposition of memory management using global memory buffers and local pointer buffers [32]. Falk et al. present an approach for quasi-static scheduling of dynamic dataflow programs that is based on strategically clustering static dataflow graphs from within dynamic graphs in which they are embedded [11]. Liu et al. present a novel workflow for software synthesis of synchronous dataflow graphs

that is based on a concept of dynamic single appearance schedules [28].

Compared to prior works on software synthesis, such as those described above, a key distinguishing characteristic of our contribution is the general and formal modeling of schedules, and the incorporation of the schedule as an input to the synthesis process. This characteristic makes our approach largely complementary to the works discussed above, making it possible to envision the approach as a unifying back-end for different types of approaches for scheduling and synthesis. For example, DSGs could be applied to model the class of dynamic single appearance schedules proposed by Liu, and the resulting models could be connected to the software synthesis techniques presented in this paper. Exploring these complementary relationships represents a useful direction for future work.

4 Design flow

In this section, we introduce a specific design flow to which our proposed new software synthesis techniques can be applied, and we discuss implementation issues related to RAs, SCAs, DSG edges, and dataflow graph execution that we have addressed in developing this design flow. Building on the semantics of DSGs that were introduced in [47], this section presents new features in the DIF and LIDE Packages that have been developed to support the prototyping of design flows that employ DSG models.

Figure 2 illustrates the DSG-integrated design flow that we investigate in this paper. In this approach, the user specifies (a) a dataflow model of an application (*application graph*), and (b) a DSG model (*schedule graph*) of a schedule for the application graph. These two graphs are specified using the DIF Language, which we introduced in Sect. 2.1.

In addition to the application graph and schedule graph, one or more actor libraries are assumed to be available. These libraries provide the internal implementations of the individual actors that are referenced in the application and schedule graphs. The libraries can in general be

Fig. 2 DSG-integrated design flow developed around the DIF-DSG tool for software synthesis

developed in terms of one or more actor implementation languages (*target languages*). In this paper, we demonstrate the design flow and software synthesis tool illustrated in Fig. 2 using C as the target language.

DIF-DSG currently supports actor libraries that are developed using LIDE (see Sect. 3). The multicore-targeted capabilities of DIF-DSG presented in this paper currently work with actor libraries that are developed using LIDE-C, which provides C implementations of the LIDE APIs. Extension of the methods and tools developed in this paper to other target languages, including hardware description languages, is an interesting direction for future work.

Our software synthesis tool, called *DIF-DSG*, receives the application graph and DSG as input, and generates a software implementation of the DSG together with the application graph in the target language. The software synthesis process assumes that implementations of the application graph actors in the target language are available from the libraries described above. These actor implementations are instantiated in the generated code. Configuration of and communication between these actors is fully coordinated in the generated code along with execution of the designer-specified schedule. DIF-DSG is represented in Fig. 2 by the large shaded rectangle in the figure. We describe details of software synthesis in DIF-DSG further in Sect. 6.

4.1 RA implementation

As described in Sect. 2, RAs can be viewed as wrappers for firing specific application graph actors, and firing of an RA A in a DSG G corresponds to a guarded firing of the referenced actor $ref(A)$ in the associated application graph. More specifically, the following steps are involved in the execution of an RA.

- *Pre-processing* The pre-processing stage of an RA firing is carried out by the optional `pre` function that may be associated with the RA. If the `pre` function is not defined for an RA, then its pre-processing stage is skipped. A `pre` function, when it is defined in DIF-DSG, is provided as a function pointer argument to the function that initializes an RA.
- *Referenced actor invocation* When an RA is initialized, it is associated with a unique application graph actor as the referenced actor of the RA. A guarded invocation of the referenced actor is carried out in this core step of the RA execution process. The guard in this context corresponds to the result of the lightweight dataflow `enable` function for the associated actor (e.g., see [27]). As mentioned in Sect. 2.5, evaluation of the guard can be bypassed if it is known by some form of static or dynamic analysis that it will always be `true`. For

example, when executing DSG-based models of static schedules for synchronous dataflow (SDF) graphs, it is possible to bypass guard checking in RAs if appropriate methods are used in the construction of the underlying schedules.

- *Post-processing* In a manner similar to the pre-processing stage, the post-processing stage of an RA firing is carried out by the optional `post` function that may be associated with the RA. The post-processing stage is skipped if no `post` function is associated with the RA. As with the `pre` function, when a `post` function is defined, it is provided as a function pointer argument in the initialization of the corresponding RA. One possible use of a `post` function is in the derivation of a value that is to be encapsulated in the DSG token that is produced on the output of the RA. If no such value is provided by the `post` function, then the DSG token that is produced can be viewed as having a null value associated with it. In this case, the token helps to direct the flow of control through the DSG even though its value is not relevant.

4.2 SCA implementation

Schedule control actors (SCAs) in DSGs provide a general mechanism to direct the flow of DSG tokens in DSGs. Since DSG tokens serve to enable actors that receive these tokens at their inputs, the direction of such tokens can be viewed as influencing the order in which actors within a DSG are executed. Currently, four specific SCA types are provided in DIF. This library of SCAs can readily be extended to include other SCAs that are based on the general “SCA design rules” defined in [47]. The four types of SCAs that are currently available in DIF are summarized as follows.

- *Loop SCAs* There are two types of loop SCAs in DIF—static and dynamic loop SCAs. Loop SCAs are used to direct DSG tokens through a specific output port, called the *body port*, for some number of successive iterations before the next DSG token output by the actor is directed through a second output port. Tokens on the body port can be viewed as enabling the body of a loop, while the second output port can be viewed as being associated with the part of the DSG that follows the loop. For further details on this type of SCA, we refer the reader to [47].

A static loop SCA is provided an iteration count as a static (compile time) parameter. On the other hand, a dynamic loop SCA receives iteration counts from DSG tokens that it consumes on one of its input ports. Thus, the iteration counts of dynamic loop SCAs can vary at

run-time, based on manipulations to the values of the DSG tokens that are provided to them.

A *loop-and-exit* (LAE) SCA is a variation of the static loop SCA. This actor has only one output port, which is its body port. The actor produces outputs on its body port for a pre-defined number of iterations. After its iteration count is exhausted, the SCA jumps out of (exits) the SDSG to any remaining “wrapup” or continuation code at the end of the enclosing thread. The LAE SCA can be viewed as a special SCA that incorporates control of the overall enclosing SDSG.

- **Conditional SCAs** There are two types of conditional SCAs in DIF—the *if* SCA and the *fi* SCA. Intuitively, an *if* SCA is used to route its output DSG token to one of two output ports, thereby enabling one of two different DSG actors for subsequent execution in the schedule graph. Conversely, the *fi* SCA consumes an input DSG token from one of two input ports, and produces output on a single output port. Thus, a common next actor in the DSG will be enabled regardless of which input the input DSG token came from.

4.3 SDSG edges

The DSG edges in LIDE-C are optimized to exploit the global token population property. In particular, each SDSG edge in LIDE-C is implemented as a FIFO with unit size. Furthermore, we apply a streamlined FIFO abstract data type (ADT) implementation from LIDE that is designed specifically for the case where the buffer size is 1.

4.4 Execution of DSGs

DIF-DSG provides capabilities to synthesize a C implementation of a DSG that is specified in the DIF Language. We refer to this synthesized C implementation as the “synthesized implementation” of the associated DSG specification. The synthesized implementation of a DSG is encapsulated within an instance of the LIDE-C DSG ADT.

In general, to execute its associated DSG, a *DSG scheduler* needs to be applied to a DSG ADT instance. By a DSG scheduler, we mean a software module that takes as input a DSG, and executes the actors in a manner that preserves DSG semantics—e.g., by firing DSG actors only when they become enabled by the presence of DSG tokens at their inputs. Since an SDSG always has at most one DSG token during its execution, there is always a unique actor that is to be fired next, once the current firing completes. A DSG scheduler provides a mechanism to carry out this and other aspects of DSG semantics. When the “DSG” qualification is understood from context, we may sometimes write “scheduler” in place of “DSG scheduler”.

Since DSGs are themselves dataflow graphs, they can be executed by any dataflow scheduler that supports the variant of core functional dataflow (CFDF) semantics that DSGs adhere to. Thus, a scheduler can be used as a DSG scheduler even if it is not specialized to the semantics of DSGs. For details on the relationship between DSGs and CFDF semantics, we refer the reader to [47], and for details on the CFDF model of computation, we refer the reader to [35].

DSG schedulers are discussed further in Sect. 7.

5 Concurrent DSGs

Mapping of dataflow applications onto multicore platforms involves significant complexity, which can greatly increase development time. A major part of this complexity comes from scheduling actors across the available cores, and managing communication and synchronization across the cores. To help manage this complexity, concurrent DSGs (CDSGs) raise the level of abstraction for working with dataflow schedules. In this section, we discuss support for CDSGs in DIF-DSG.

In the current version of DIF-DSG, and in the remainder of this paper, we assume that each actor of a given application graph is mapped to a single thread. We refer to this as the *Single Thread per Application Actor* (STAA) assumption. Generalizing the methods of this paper and the features in DIF-DSG to relax the STAA assumption is a useful direction for future work.

5.1 Inter-SDSG coordination actors

A CDSG is composed of multiple sequential DSGs, where communication between different sequential DSGs is carried out through a special class of DSG actors that are devoted to communication and synchronization between different SDSGs. Specifically, multiple SDSGs can be integrated into a CDSG through a class of actors called *Inter-SDSG Coordination Actors* (ISCAs). ISCAs can be viewed as a generalization of the interprocessor communication actors that were introduced in [47]. For example, ISCAs also include actors for inter-thread communication, where SDSGs are mapped to separate threads, and the threads involved can be assigned to the same processor or to different processors.

Except for the edges that are directed between ISCAs, each graph element (actor or edge) in a CDSG is contained within one of the SDSGs that make up the CDSG. Given a CDSG G_c and an actor a in G_c , the unique SDSG that contains a is denoted as $sdsg(a)$.

The set of SDSGs contained within a given CDSG G_c is referred to as the *SDSG set* of G_c , and denoted as $\sigma(G_c)$.

Given a CDSG for an application graph G , parallel execution of actors in G is achieved when multiple RAs in multiple members of $\sigma(G_c)$ are executed at the same time.

CDSG-based software synthesis in DIF-DSG is currently targeted to sequential and multicore implementations that employ C/Pthreads programs as the output of software synthesis. Thus, DIF-DSG currently supports C/Pthreads-based implementation of ISCA. However, due to their model-based orientation in terms of DSG semantics, these actors can be retargeted in different ways for different platforms and target languages. For example, specialized ISCA can be developed to provide communication across an AXI interface, and thereby support DSG-based schedule implementation that involves offloading tasks from processor cores to hardware accelerators. In such a scenario, one type of ISCA would be used for communication between different processor cores and another type for communication across the AXI interface. Abstracting the scheduling process for this class of heterogeneous platforms may be useful to investigate trade-offs that involve different AXI interface options [43].

In the remainder of this section, we describe the implementation of two ISCA, the *snd* and *rec* actors, in DIF-DSG. As mentioned in Sect. 2, these actors are used to send and receive data across different SDSGs.

In the current version of DIF-DSG, we assume that each *snd* actor a_s has a corresponding *rec* actor a_r , where a_r corresponds uniquely to a_s . This assumption may be relaxed in future versions of DIF-DSG and other implementations of CDSGs (e.g., a single *snd* actor may send data to multiple *rec* actors to achieve broadcasting functionality). In the enclosing CDSGs, an edge is inserted from each *snd* actor a_s to its corresponding *rec* actor to model the associated inter-thread data transfer.

CDSG actors a_s and a_r are used to communicate data from one application graph actor X_s to another application graph actor X_r , where the actors X_s and X_r are contained in different SDSGs $sdsg(a_s)$ and $sdsg(a_r)$, respectively. The actors X_s and X_r communicate through a single FIFO buffer b_{sr} , which can be accessed directly by these actors using Pthreads APIs.

Figure 3 illustrates the input and output edges of the *snd* and *rec* actors. The detailed structure and operation of these actors is described as follows.

- *rec* The actor a_r has one input edge e_{ri} and one output edge e_{ro} , which are both contained within $sdsg(a_r)$. There is a second input edge to a_r , which is directed from a_s , and which we typically depict in drawings as a dashed edge. We refer to this type of edge as an *inter-thread communication edge* in the SDSG. This dashed edge represents the coupling between a_r and a_s as corresponding ISCA. Consideration of this type of edge

Fig. 3 An illustration of *snd* and *rec* actors in CDSGs

can be relevant to certain kinds of dataflow analysis, such as analysis of synchronization structure, buffer memory requirements, and throughput (e.g., see [44]).

On each firing, a *rec* actor a_r waits, using the blocking API function of Pthreads called `pthread_cond_wait`, until the population of b_{sr} exceeds 0. At this point, the *rec* actor finishes execution, and a DSG token is produced on its output to enable the actor in $sdsg(a_r)$ at the sink of e_{ro} . In general, one or more *rec* actors can be used to ensure that each firing of an application graph actor A has sufficient data from other threads that produce input data for A .

- *snd* Like a_r , the actor a_s has one input edge and one output edge that reside within the same SDSG. We denote these edges, respectively, as e_{si} and e_{so} . As explained above, there is a second output edge of a_s , which is directed to a_r , and is called an inter-thread communication edge. The delay on the dashed edge is set to the delay of the corresponding application graph edge (X_s, X_r) . On each firing, a *snd* actor a_s waits, using `pthread_cond_wait`, until the population of the buffer b_{sr} is less than its buffer capacity. Once at least one unit of free space has been validated in the buffer, a_s finishes execution, and a DSG token is produced on its output to enable the actor in $sdsg(a_s)$ at the sink of e_{so} . In general, one or more *snd* actors can be used to ensure that each firing of an application graph actor B has sufficient empty space on its output ports to produce any data needed from B by actors that are assigned to other threads.

5.2 Delays

In dataflow representations of signal processing applications, *delays* on edges generally correspond to initial tokens on buffers associated with the edges (e.g., see [1]). Given a dataflow graph edge e , we denote the magnitude of the delay (number of initial tokens) on e as $delay(e)$. We distinguish between two different types of delays when working with CDSGs.

1. Application delays. Delays on an application graph edge e_{app} are implemented as initial tokens on the application graph buffer associated with e_{app} . Since

intra-thread communication is not modeled with edges in DSGs, an application graph delay does not show up in the CDSG when the source and sink vertices of e_{app} are assigned to the same thread. If the source and sink of e_{app} are mapped to different threads, then in general there will be a set S of inter-thread communication edges in the CDSG corresponding to e_{app} .

From the STAA assumption, we are guaranteed that the set S always has exactly one element e_{itc} (when the source and sink of e_{app} are mapped to separate threads). The delay on this DSG edge is set to the delay of the corresponding application graph edge: $delay(e_{itc}) = delay(e_{app})$.

2. Intra-thread delays. In a CDSG, an *intra-thread edge* is an edge whose source and sink vertices are contained in the same SDSG. All of the intra-thread edges in a given SDSG are assigned zero delay, except for a single intra-thread edge e , called the *starting edge* of the SDSG. The starting edge e is assigned $delay(e) = 1$. When a CDSG starts executing, the execution on each SDSG V commences with the actor at the sink of the starting edge of V .

Note that this approach to assigning intra-thread delays is consistent with the global token population property of SDSGs, which was reviewed in Sect. 2.

5.3 Example

To illustrate the concepts and constructions developed in this section, Fig. 4 shows a simple example of an application graph and an associated CDSG for a 2-thread schedule. The application graph, shown in Fig. 4a, consists only of homogeneous SDF actors, except for the actor labeled SW, which is a Boolean dataflow (BDF) *switch* actor. The

switch actor is a fundamental BDF actor. For details on its functionality and its modeling in SDSGs, we refer the reader to the works by Buck [8] and Wu [47], respectively. Here, by *homogeneous SDF*, we refer to a special case of SDF in which the production and consumption rates on actor ports are identically equal to 1 [22].

In Fig. 4b, each actor labeled R_k represents a reference actor $ref(X_k)$ for application graph actor X_k . In addition to these reference actors, the CDSG in Fig. 4b also contains four SCAs—one each of the *if*, *fi*, *snd*, and *rec* SCAs. Actor X_5 provides the control input and actor X_1 provides the data input to the switch actor in the application graph. The actors mapped to Thread 2 are executed when a *true*-valued control token is consumed by the switch actor, whereas *false*-valued control tokens drive the execution of X_3 through its reference actor R_3 .

In Fig. 4b, the starting edges for the two SDSGs are (fi, R_1) and (R_4, rec) . Each of the “D” annotations on these edges represents a unit delay, which corresponds to the initial position of the DSG token for the associated SDSG.

6 Software synthesis

In this section, we introduce the software synthesis capabilities of DIF-DSG and describe how the intermediate representations of DIF are applied to support these capabilities. We also provide more details about the workflow for applying software synthesis using DIF-DSG. The DIF-DSG framework takes as input an application graph and a schedule graph (DSG). As discussed in Sect. 4, each actor in the application graph is an instantiation of an actor from a library of LIDE-C actors. These actors can in general be a mix of actors that are taken from the built-in actors

Fig. 4 An example that illustrates the concepts and constructions developed in this section

within LIDE-C, and user-defined LIDE-C actors that are constructed using the APIs and utilities provided in LIDE-C.

Application graphs and schedule graphs are specified as input to DIF-DSG using the Dataflow Interchange Format (DIF) Language [15, 27]. The DIF Language is designed for specification of abstract dataflow graph models for DSP systems. Here, by “abstract”, we mean that the topology (vertices and edges) of the models, along with relevant dataflow properties, and parameters of actors (vertices) and edges are specified, while the internal functionality of the actors and edges is *not* included as part of the specifications. For software synthesis, these internal specifications can be provided by external libraries, such as LIDE-C. DIF supports various forms of dataflow, including SDF [22], cyclo-static dataflow [5], Boolean dataflow [8], CFDF [35], and several others.

The DIF parser is used to construct intermediate representations, based on data structures within the DIF Package, for the specified application graph and schedule graph. These representations are jointly processed in DIF-DSG to produce as output a single- or multi-threaded implementation of the given application graph together with the given schedule graph. A multi-threaded implementation is generated whenever the input schedule graph contains multiple SDSGs. Run-time support for executing the generated schedule efficiently in terms of DSG semantics is also generated as part of the software synthesis process. The output of DIF-DSG is therefore in the form of a single- or multi-threaded C program, which can be compiled onto specific target platforms using the platform-based compilers associated with those platforms. As mentioned in Sect. 5, multi-threaded implementations generated by DIF-DSG use Pthreads as the underlying threading API.

For more details about the DIF Language, we refer the reader to [27]. Certain DSG-specific details must be included in DIF Language specifications of DSGs. These details include the referenced actor associated with each RA; iteration counts associated with loop-related SCAs; and the names of the `pre` and `post` functions that are associated with the RAs. These details are not covered here for brevity, and because they are primarily matters of syntax; the associated details can be attached to schedule graph specifications in many other ways, depending, for example, on syntactic conventions of languages that are being extended with these features.

Note that the DIF Language specification of an RA does not necessarily need to have a `pre` function or `post` function—either or both of these functions can be omitted for a given RA. Specification of these functions is therefore optional in the DIF specification for an RA. In the absence of a `pre` or `post` specification in a DIF program, it is assumed that the associated RA does not have

such a function associated with it or equivalently, that the function exists, but simply does nothing (a “no-operation” function).

6.1 Internal representations for dataflow constructs in DIF-DSG

As described previously, application graphs and schedule graphs specified using the DIF Language are converted into internal representations within the DIF-DSG tool. These representations, based on the dataflow graph classes of DIF, are utilized for synthesis and optimization of the generated software. Principal aspects of the information stored within these intermediate representations include:

- Graphical connectivity between actors and edges in the application graph and schedule graph.
- Relevant details about actors in both graphs.
- Relevant details about edges in both graphs.

For example, application graph actors within the DIF-based intermediate representation include the following fields:

- *Instance name* the name of each specific actor instance.
- *Type name* the actor type from which the instance is derived.
- Details about input and output ports.

Similarly, examples of fields for schedule graph actors include:

- Instance name.
- Type name (e.g., to identify what kind of SCA each SCA instance is associated with).
- Referenced actor (for RAs).
- Details about input and output ports.

6.2 Code generation

The DIF-DSG framework generates well-structured, human readable code in LIDE-C format. DIF-DSG divides the generated code into different components (ADTs) for the application and schedule graphs, which enhances the modularity of the derived implementation. For example, a common application graph implementation can be integrated efficiently with different schedule implementations to experiment with alternative schedules for the application.

The software synthesized by DIF-DSG includes two `.c` files—one for the application graph and the other for the schedule graph. A DSG implementation that is synthesized

by DIF-DSG can be executed using different DSG schedulers—that is, different methods for interpreting the synthesized DSG implementation (see Sect. 4.4). Two alternative schedulers that we use in our experiments are discussed in Sect. 7. The set of supported schedulers for this purpose can easily be extended as more scheduler techniques are incorporated into DIF-DSG.

7 DSG schedulers

To execute its associated DSG, a *DSG scheduler* can be applied to a DSG ADT instance. By a DSG scheduler, we mean a software module that takes as input a DSG, and executes the actors in a manner that preserves DSG semantics—e.g., by firing DSG actors only when they become enabled by the presence of DSG tokens at their inputs. Since an SDSG always has at most one DSG token during its execution, there is always a unique actor that is to be fired next within an SDSG once the current firing completes. A DSG scheduler provides a mechanism to carry out this and other aspects of DSG semantics. When the “DSG” qualification is understood from context, we may sometimes write “scheduler” in place of “DSG scheduler”.

Since DSGs are themselves dataflow graphs, they can be executed by any data flow scheduler that supports the variant of core functional dataflow (CFDF) semantics that DSGs adhere to. Thus, a scheduler can be used as a DSG scheduler even if it is not specialized to the semantics of DSGs. For details on the relationship between DSGs and CFDF semantics, we refer the reader to [47], and for details on the CFDF model of computation, we refer the reader to [27, 35].

On a multi-threaded implementation target, a CDSG can be scheduled by mapping each SDSG to a separate thread and using an *SDSG scheduler* for each of the resulting threads. An SDSG scheduler is simply a DSG scheduler that is specialized to SDSGs (i.e., it handles only SDSGs). This is how CDSGs are scheduled in DIF-DSG. In this scheduling approach, communication and synchronization between threads is handled entirely by `snd` and `rec` SCAs.

In Sects. 7.1 and 7.3, we present two SDSG schedulers that are implemented in DIF-DSG.

7.1 Simple scheduler

A general scheduler that we apply in our experiments as a baseline is an adaptation of the *simple scheduler* that is provided in LIDE [41]. The simple scheduler in LIDE applies a form of scheduling called *canonical scheduling* of CFDF graphs [36]. Canonical scheduling involves ordering the actors at compile time, and traversing actors according

to the order at run-time. When a given actor is visited during the traversal, it is fired if it is enabled, otherwise it is skipped during the current iteration of the traversal. After the last actor in the ordering is visited, the traversal repeats, starting again with the first actor. The simple scheduler therefore provides a form of round-robin scheduling. Traversals through the actor ordering are carried out repeatedly until some stopping condition for the application is met or the schedule is interrupted or stopped through some external means. This scheduling technique and the LIDE simple scheduler can be adapted easily to handle DSGs even though DSGs deviate from CFDF semantics in some respects.

While the simple scheduler is not efficient, its underlying round-robin method is very general, and because of its generality, it can be applied directly to DSGs. Therefore, it is an off-the-shelf scheduling approach that is useful as a baseline when evaluating more sophisticated DSG scheduler strategies.

Advantages of the simple scheduler include its simplicity and generality, which make it useful, for example, during functional validation and rapid prototyping. Its main drawback is that it can be highly inefficient. Thus, an investigation of efficient scheduling techniques should not be limited to use of the simple scheduler. In the remainder of this paper, when we write “simple scheduler”, we mean (unless otherwise stated) the adapted version of the LIDE simple scheduler that we use in DIF.

7.2 Experimentation with DSG schedulers

The model-based methodology for schedule design and synthesis developed in this paper introduces the problem of scheduling DSG-based schedule representations. Such representations incorporate pre-specified assignments of actors to processing resources (e.g., threads), and pre-specified logic for time-multiplexing actors that are mapped to the same resource. In contrast, existing scheduling methods for related forms of dataflow are focused primarily on deriving assignments and orderings of actors, which is a very different scheduling problem from that of scheduling DSGs. Indeed, “DSG scheduler design” is a novel synthesis sub-problem that is introduced by this work. The simple scheduler is applied as a first baseline for experimental comparisons in the context of this new scheduling problem.

7.3 DSG token tracking

In this section, we present an approach for SDSG scheduler optimization that we refer to as *DSG token tracking* (DTT). DTT exploits the global token population property of SDSGs. Recall that this property ensures that at any given


```

repeat
  Invoke the current SDSG actor  $C$ 
  Record the index  $I$  of the edge on which  $C$  has produced its output token
   $C = \text{sinks}[I]$ 
until  $\text{terminate}() = \text{true}$ 

```

Fig. 5 A pseudocode representation the DTT approach for SDSG scheduler optimization

time during execution of a properly constructed SDSG, there is at most one DSG token in the graph. Thus, once the firing of an SDSG actor completes, we need only to determine the edge e on which the firing produces its output DSG token. The sink actor of this edge e is the unique actor that needs to be executed as the next firing in the interpretation of the enclosing SDSG.

DTT utilizes these concepts, and is based on efficiently tracking DSG tokens as they are produced during interpretation of a DSG. Here, by “tracking”, we simply mean determining the edge on which the associated token resides.

The DTT approach can be summarized by the pseudocode-style representation shown in Fig. 5. Here, each iteration of the loop executes a single scheduling step, where each scheduling step is responsible for carrying out a single firing of an SDSG actor and determining the next SDSG actor that is to be fired. It is assumed that the “current SDSG actor C ” is initialized to be the first actor that is to be executed when interpretation of the SDSG is launched, as determined by the starting edge of the SDSG. Similarly, it is assumed that there is some termination condition $\text{terminate}()$ that can be checked to determine when execution of the SDSG is complete. Alternatively, the scheduler can be enclosed within an infinite loop that is executed indefinitely and may be terminated asynchronously through some kind of interrupt mechanism.

To support our implementation of the DTT approach, as represented in Fig. 5, we associate (in the synthesized code) a unique index in the range $\{0, 1, \dots, (N - 1)\}$ with each DSG edge, where N is the total number of DSG edges. We also synthesize a table (array) $\text{sinks}[I]$ that maps these “edge indices” into unique indices that are associated with the DSG actors. Specifically, $\text{sinks}[I]$ gives the index of the sink actor that is associated with the DSG edge whose index is I .

DTT is streamlined based on specialized characteristics of DSGs, including the global token population property, as mentioned above. DTT is not designed to operate directly on an application graph, but rather to be applied to a DSG.

Consider the example of Fig. 6, and suppose that the desired scheduling behavior is to execute the application graph actors in the sequence $AABBBB$. Using a round robin scheduling approach, such as that provided by the simple scheduler, the actors in the DSG would be checked

Fig. 6 An example used to illustrate DTT as a form of streamlined DSG scheduler. Parts of the DSG that are used to execute application graph actors C and D are not shown in this illustration since they are not relevant to the example. The static iteration counts associated with the two loop SCAs are annotated with the *iter* labels

iteratively at each scheduling step until an enabled actor is found. For example, after executing loop1 in Fig. 6, the next SDSG1 actor to execute is either R_A or loop1, depending on the state of loop1. A round robin scheduler may try (test for being enabled) any number of other actors in the SDSG until it finds the next actor to execute. Even if the search is restricted to successor actors of loop1, a round robin scheduler will in general need to check both SND_A and R_A . In contrast, a DTT scheduler, can move immediately to the next actor to execute (either SND_A or R_A in this case) without any testing of whether actors are enabled.

To summarize this improvement in more abstract terms: round robin scheduling involves $O(n)$ complexity at each scheduling step, where n is the number of actors in the enclosing SDSG, whereas DTT eliminates the need to test enable conditions, and involves $O(1)$ complexity at each step.

7.4 Interpreted execution versus direct synthesis

Further improvements in efficiency may be possible through methods that generate code for directly executing DSGs. Such methods, for example, might have code for SCAs generated that is followed by appropriate branching code, where this branching code is determined based on the type of SCA being used, and the particular output port on which the DSG token is produced during a given firing. Investigation of such *direct synthesis* methods for DSGs is a useful direction for future work.

Our interpreted approach for DSG execution has certain advantages, as well. These advantages relate to general differences between interpreted and compiled execution of computer programs. When DSGs are interpreted, only the DSG itself needs to be synthesized, while the logic for

executing the DSG can be re-used through pre-defined runtime library components, or through a separately provided scheduler. Similarly, interpreted DSGs facilitate efficient *just-in-time scheduling* techniques where DSGs may be constructed, adapted, or input at run-time, and then executed immediately after they become available. Prior work on just-in-time scheduling of Parameterized and Interfaced Synchronous DataFlow (PiSDF) graphs has shown promising results [14]. An interesting direction for future work is the study of just-time-time techniques for dataflow graphs that utilize the concepts of DSGs, DSG schedulers, and interpreted DSG execution.

7.5 Soundness

The soundness of the proposed approach to DSG-based scheduler design and software synthesis is inherited from fundamental properties of determinacy that have been established for dataflow process networks [23]. In particular, dataflow process networks are shown to provide determinate execution when actor firing rules satisfy a technical concept of sequentiality. The core functional dataflow (CFDF) model of computation upon which DIF-DSG is based (see Sect. 2.3) provides sequential firing rules, and therefore ensures determinacy based on the aforementioned results established for the more general dataflow process network model. Moreover, in the proposed approach to applying DSGs, actors are fired only when they are enabled. This requirement is met either through the use of guards associated with RAs or by static or dynamic analysis that permits the bypassing of guards (see Sect. 2.5). The constraint that only enabled actors are fired together with the relationship of CFDF to dataflow process networks are important foundations that ensure the soundness of the methods proposed in this paper.

8 Case study: real-time classification system

In this section, we demonstrate the utility of DIF-DSG and its underlying modeling techniques through a case study involving a real time classification system. The classification system is designed to discriminate among people, vehicles, and noise using acoustic and seismic signals. Here, by “noise”, we mean the absence of any person or vehicle.

For details on the algorithms and applications associated with this classification system, we refer the reader to [25, 26]. The case study in this paper goes beyond the simulation and prototyping experiments reported in [25, 26] in its use of DSGs and DIF-DSG as central parts of the design and implementation process. The experiments described in [25, 26] are carried using hand-implemented dataflow graphs that do not employ DSG modeling nor the DIF-DSG software synthesis tool. However, these earlier studies provide useful LIDE-C actor implementations, which we apply in this case study.

The embedded processing platform used in our experiments is the Raspberry Pi 3 Model B, which is equipped with 1 GB RAM, a 4× ARM Cortex A53 CPU, and a Broadcom VideoCore IV GPU. In the experiments described in this paper, we do not use the GPU.

The input to our classifier application consists of a stream of acoustic data and a stream of seismic data that are delivered from two sensors through an A/D converter. In practice, the data is obtained directly from the sensors in this way. However, we compare the run-time and memory requirements for different implementations using the same input data. For this purpose of reproducibility, pre-collected input frames (frames of acoustic and seismic signals) are used in our experiments. Details about these datasets can be found in [30].

8.1 Application graph

In the classifier application presented in [26], four different modes of operation are provided for people/vehicle/noise classification. Among these modes, we employ in our experiments only the ALFFS (Accumulation of Local Feature-level Fusion Scores) mode. The ALFFS approach provides superior accuracy, although this enhanced discrimination capability comes at the expense of longer execution time.

The application graph for our ALFFS application is illustrated in Fig. 7. The production and consumption rates associated with actor ports are annotated next to the ports. These rates are expressed in terms of three static application parameters— D , W , and F . These parameters, respectively, represent the frame size, number of windows per frame, and number of features extracted per window. For more details about these parameters along with the associated notions of frames and windows in the ALFFS

Fig. 7 Dataflow graph for ALFFS application

application, we refer the reader to [25, 26]. In our experiments, we use $D = 4096$ samples per frame, $W = 50$ windows, and $F = 50$ features.

Descriptions of the actors in Fig. 7 are summarized briefly as follows.

- **File Source and File Sink** These actors represent interfaces for reading and writing, respectively, pre-collected sensor data that is stored on a microSD card, which is attached to the targeted Raspberry Pi platform.
- **Feature Extraction** There are two instances of this actor, one for handling acoustic data and the other for seismic data. These actors apply cepstral analysis to extract features from digitized sensor data. For FFT computation, which is an important part of cepstral analysis, we employ a LIDE-C wrapper around an optimized module from the FFTW library [12]. The generated cepstral coefficients are subsequently employed (in the downstream portion of the dataflow graph) as the features of the input frame.
- **Feature Concatenation** The coefficients extracted from each pair of corresponding windows associated with the two sensing modalities are concatenated by this actor before arriving as input to the SVM (support vector machine) Bank actor.
- **SVM Bank** The cepstral features extracted from the Feature Extraction actors are sent as input to the SVM bank actor. In a given firing of the SVM Bank actor, the actor accesses trained parameters that are stored in memory as static parameters of the actor. Three classification scores are generated by the SVM bank actor, one for each type of pairwise discrimination a versus b , where $a, b \in \{N, P, V\}$, and N, P, V here represent, respectively, the noise, person, and vehicle classes. For more details on the algorithm underlying the SVM Bank actor, we refer the reader to [25].
- **ALFFS Score** The ALFFS Score actor takes as input W blocks of M values each, where each block corresponds

to the scores derived from a specific window in the current input frame. Corresponding elements of these blocks (triples in our case, since $M = 3$) are added in the ALFFS Score actor, which results in a single, accumulated score for each class.

- **Threshold Actor** This actor consumes a block of real values $x_1, x_2, \dots, x_M$, and simply applies a threshold τ to each one to produce a sequence of binary numbers $y_1, y_2, \dots, y_M$. Each binary number represents a pairwise classification result, which is used as an intermediate result in the overall multiclass classification operation that is performed by each dataflow graph iteration. In this application, the block size and threshold are configured as $M = 3$ and $\tau = 0$.
- **Multiclass Decision Actor** Each firing of this actor combines a block of M pairwise classification results into a single multiclass classification result. The actor applies a simple voting rule to generate a single classification result from within the set $\{N, P, V\}$. In case of a tie (all three input predictions are different), the actor produces N (noise) as the classification result.

Further details about these actors and the overall classification application can be found in [26].

8.2 Alternative DSGs

For the application graph shown in Fig. 7, we provide experimental results using three different DSGs—an SDSG, a CDSG using 2 threads, and a CDSG using 3 threads. These three DSGs are all derived by hand using the structure of the application graph and knowledge of profiled actor execution times to guide the design of the schedules. The three schedule graphs are shown in Figs. 8, 9, and 10, respectively. The actors labeled `loop1`, `loop2`, ..., `loop9` in the figures are static loop SCAs, and the actors labeled `LAE1`, `LAE2`, ..., `LAE6` are loop-and-exit actors (see Sect. 4.2).

Fig. 8 An SDSG for the application graph of Fig. 7

Fig. 9 CDSG (2 threads) for the application graph of Fig. 7

Fig. 10 CDSG (3 threads) for the application graph of Fig. 7 **Table 1** Comparison of run-times and peak memory requirements

	Avg. run-time (std) (s)	Peak memory usage (MBytes)
Hand-written	7.86 (std: 1.02×10^{-1})	7.68
Hand-written (2thread)	7.14 (std: 1.12×10^{-1})	7.70
Hand-written (3thread)	6.32 (std: 1.27×10^{-1})	7.82
SS	10.8 (std: 2.79×10^{-2})	7.78
SS (2thread)	8.87 (std: 1.08×10^{-1})	7.79
SS (3thread)	8.12 (std: 1.08×10^{-1})	7.91
SDSG-SS	10.0 (std: 1.41×10^{-2})	7.86
CDSG-SS (2thread)	9.43 (std: 2.91×10^{-2})	7.94
CDSG-SS (3thread)	6.89 (std: 5.34×10^{-3})	8.00
SDSG-DTT	8.62 (std: 1.42×10^{-3})	7.88
CDSG-DTT (2thread)	7.73 (std: 2.85×10^{-3})	7.96
CDSG-DTT (3thread)	6.50 (std: 3.02×10^{-3})	8.02

Figures 7 and 9 correspond to two- and three-stage pipelined schedules, respectively, where different iterations of the application graph execute concurrently.

8.3 Performance evaluation

For each of the three schedule graphs illustrated in Figs. 8, 9, and 10, we used DIF-DSG to synthesize an implementation. We then executed each of the resulting implementations using two different schedulers as the DSG schedulers—the simple scheduler (see Sect. 7.1) and the DTT scheduler (Sect. 7.3). This resulted in six different sets of measurements, which are represented in the lower six rows of Table 1. As a baseline for these measurements, we applied the simple scheduler directly to the application graph, without use of any DSG. This configuration is represented by the fourth through sixth rows of data (labeled “SS”, “SS (2thread)”, and “SS (3thread)”) in Table 1.

As an additional point of comparison, we included results from three hand-written schedulers, which were developed independently of DIF-DSG. The hand-written

schedulers exploited knowledge of the SDF characteristic of the application graph to construct a static schedule within each thread (i.e., for the subset of actors assigned to each thread), and thereby eliminated the use of any run-time checking of enable conditions within the threads. In the case of the two- and three-thread schedules, the assignment of actors to threads was taken to be identical to those in the corresponding multithreaded DSG versions. In all three hand-written schedulers, the ordering of actor executions within the threads was performed directly (by invoking one actor after the other directly in the code) rather than through interpretation of any DSG. The results from the hand-written schedulers are shown in the first three rows of Table 1.

For each of the 12 scheduler/DSG configurations corresponding to the 12 rows of Table 1, we validated functional correctness by comparing to the output of MATLAB reference code for the underlying algorithm using a given set of test inputs. The classification accuracy measured from this functional evaluation was 98.4%. This level of accuracy is maintained for all of the implementation alternatives investigated in this section since the alternatives all maintain the same algorithmic configurations for feature extraction and classification.

Table 1 summarizes measurements of run-times and peak memory usage for the 12 different configurations that we experimented with. The run-times reported here are the average times for processing 100 frames of bimodal data, where the average is taken over 200 test cases. Standard deviations for these measurements are also reported in the table (with the abbreviation “std”). One frame of data contains 1 second of acoustic and seismic data sampled at 4096 Hz.

As expected, the two sequential (single-thread) configurations that employ the simple scheduler are the slowest, since they involve repeatedly visiting actors and checking enable conditions. For both of these configurations, the traversal orders for the simple scheduler are constructed carefully (by hand) to help minimize the rate at which enable checks are found to fail at run-time. This is a useful

design-time optimization (for some applications) that can be performed heuristically using topological sort analysis before deploying an implementation that uses the simple scheduler. The SS-based scheduling process is different for the SS and SDSG configurations in that in the former, the enable conditions for application graph actors are checked at run-time, whereas in the latter, the enable conditions for DSG actors are checked.

As the results in Table 1 show, the CDSG-based configurations generally improve performance by making use of multiple threads. This performance improvement is achieved in a structured way based on dataflow semantics, which avoids pitfalls associated with unstructured use of threads (e.g., see [20]).

The performance improvement due to use of multiple threads is significant for both the SS and DTT configurations of SDSGs and CDSGs. We expect that the improvement is less than the ideal 2X and 3X speedups (for two and three threads) (a) due to overheads that are incurred due to inter-thread communication and synchronization, and (b) because the SVM Bank actor is a bottleneck actor in the application design, which limits the available parallelism. Future work on the application graph to decompose this bottleneck (into multiple actors) is a useful direction to expose more parallelism in the design.

Use of a DTT scheduler brings significant improvement in all cases—SDSG-DTT, CDSG (2 thread)-DTT, and CDSG (3 thread)-DTT have 13.8%, 18.0%, and 5.66% less run-time compared to SDSG-SS, CDSG (2 thread)-SS, and CDSG (3 thread)-SS, respectively.

In all three sets of experiments—sequential, two-thread, and three-thread—the hand-written schedulers are the fastest. However, the performance of the DTT schedulers comes close to that of corresponding hand-written schedulers. The performance gap between the hand-written and DSG-based schedulers can be viewed as a cost for raising the level of abstraction in the schedule representation, which is enabled by the DSG model. This performance gap is analogous in some ways to gaps in performance that are often observed between hand-optimized assembly code and assembly code that is generated from a C compiler. The problem of DSG scheduler design provides a concrete formulation to help address the performance gap associated with DSGs, and the DTT technique provides a first example of a solution to this problem.

In summary, the experiments reported here help to validate the correctness of the DIF-DSG software synthesis process; the capability of CDSGs to provide efficient, model-based representations for implementing multi-threaded signal processing systems; and the utility of DSG token tracking as a first approach to optimized design of DSG schedulers.

8.4 Lines of code comparison

By raising the level of abstraction for dataflow schedule design, DIF-DSG enables more concise representations of embedded software implementations. This can be measured as a reduction in the lines of code (LOC) that are required for the programmer to develop and maintain. More specifically, the LOC represents the total number of lines of source code in an application, with any code that is synthesized (auto-generated) excluded. LOC is used as a natural metric for evaluating the compactness of software (e.g., see [6, 37]).

In this section, we present a comparison of the LOC for DSG implementation with and without DIF-DSG. The comparison is performed for the classifier application. The comparison is focused only on scheduling code since DIF-DSG code synthesis is targeted to scheduling functionality. Although the scheduling code represents a relatively small portion of the overall application code, this code is of special importance due to its impact on performance, and its interaction with all application graph components. Therefore, significant reductions in scheduling-related LOC are useful to reduce effort and cost associated with development, experimentation and maintenance.

Figure 11 illustrates the results of the LOC comparison experiment. The results compare the DIF language LOC levels for three different schedules with the corresponding C language scheduling code that is synthesized by DIF-DSG. The “without DSG” implementations represented in Fig. 11 show the LOC levels for the C language scheduling code that is synthesized by DIF-DSG. These data points give an indication about the amount of code that must be developed and maintained to apply equivalent schedules to what DIF-DSG provides, but without the higher level of design abstraction and code synthesis capability provided by DIF-DSG. The experiment therefore provides a quantitative indication of how the higher level of abstraction offered to the designer by DIF-DSG compares in compactness to the equivalent functionality that would need to be developed and maintained as source code in the absence of DIF-DSG. From the

Fig. 11 Lines of code (LOC) comparison with and without DIF-DSG

Fig. 12 Detailed design flow

results, it is seen that DIF-DSG provides significantly better LOC efficiency for all three of the schedules examined—an SDSG, a CDSG with 2 threads, and a CDSG with 3 threads.

8.5 Design flow details

In this section, we summarize in more detail the design flow for synthesizing software from cooperating, DIF-based application graph and DSG representations. The design flow is illustrated in Fig. 12.

A given implementation using DIF-DSG involves, at the input (source code) to the software synthesis process, two DIF Language files—one file gives the application graph specification, while the other gives the DSG specification. Since the DSG representation is dataflow-based, the DIF Language is naturally suited to specifying schedules in this format.

The software synthesis process in DIF-DSG involves traversing the data structures associated with the application graph and DSG, and generating C code that constructs those graphs. Additionally, code must be generated to ensure proper linkage between reference actors in the DSG and their corresponding referenced actors in the application graph. Since the underlying application and schedule representations in DIF-DSG are model-based, the software synthesis process can readily be retargeted to generate code in other languages. Application of the methods developed in this paper to other languages and platforms is an interesting direction for further study.

9 Conclusion

This paper has presented new methods and tools for synthesis of software for signal processing systems. The novelty of the methods centers on their support for the

dataflow schedule graph (DSG) as a formal model of schedules for dataflow-based application representations. In conventional dataflow-based software synthesis techniques, schedules are represented using formal representations that are very restricted in their applicability or using general representations that are constructed in ad-hoc ways, without any formal connection to dataflow. This paper has developed software synthesis techniques that operate on the DSG model, which is both general in its applicability to a broad class of static and dynamic dataflow representations, and formal in its underpinnings in terms of dataflow semantics. The paper has presented the first development of multicore implementation techniques using DSGs, and integrated these techniques into a new software synthesis tool. The tool generates code automatically from coupled dataflow representations of applications and schedules, and incorporates an optimized run-time system that exploits special characteristics of DSGs. The new software synthesis techniques are demonstrated through experiments involving a state-of-the-art signal processing system for real-time detection of people and vehicles using acoustic and seismic sensors.

Acknowledgements This material is based upon work supported by Army Research Laboratory, National Science Foundation, and Air Force Office of Scientific Research under Grant No. FA9550-18-1-0068.

Compliance with ethical standards

Conflict of interest On behalf of all authors, Kyunghun Lee states that there is no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this

article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

Appendix

See Table 2.

Table 2 List of abbreviations

ADT	Abstract data type
ALFFS	Accumulation of local feature-level fusion scores
API	Application programming interface
CDSG	Concurrent dataflow schedule graph
CFDF	Core functional dataflow
DICE	DSPCAD integrative command line environment
DIF	Dataflow interchange format
DSG	Dataflow schedule graph
DSP	Digital signal processing
DTT	DSG token tracking
FIFO	First in, first out
GST	Generalized schedule tree
ISCA	Inter-SDSG coordination actor
LAE	Loop-and-exit
LIDE	DSPCAD lightweight dataflow environment
RA	Reference actor
SCA	Schedule control actor
SDF	Synchronous dataflow
SDSG	Sequential dataflow schedule graph
STAA	Single thread per application actor

References

- Bhattacharyya SS, Deprettere E, Leupers R, Takala J (eds) (2019) Handbook of signal processing systems, 3rd edn. Springer, Berlin
- Bhattacharyya SS, Lilius J (2018) Model-based representations for dataflow schedules. In: Principles of modeling. Springer, pp 88–105
- Bhattacharyya SS, Plishker W, Shen C, Sane N, Zaki G (2011) The DSPCAD integrative command line environment: introduction to DICE version 1.1. Technical report UMIACS-TR-2011-10, Institute for Advanced Computer Studies, University of Maryland at College Park. <http://drum.lib.umd.edu/handle/1903/11422>
- Bhattacharyya SS, Sriram S, Lee EA (1997) Optimizing synchronization in multiprocessor DSP systems. IEEE Trans Signal Process 45(6):1605–1618
- Bilsen G, Engels M, Lauwereins R, Peperstraete JA (1996) Cyclostatic dataflow. IEEE Trans Signal Process 44(2):397–408
- Boehm BW (1981) Software engineering economics. Prentice Hall, Upper Saddle River
- Buck JT (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
- Buck JT, Lee EA (1993) Scheduling dynamic dataflow graphs using the token flow model. In: Proceedings of the international conference on acoustics, speech, and signal processing
- Chen Y, Zhou H (2012) Buffer minimization in pipelined SDF scheduling on multi-core platforms. In: Proceedings of the Asia south pacific design automation conference, pp 127–132
- Damavandpeyma M, Stuijk S, Basten T, Geilen M, Corporaal H (2012) Modeling static-order schedules in synchronous dataflow graphs. In: Proceedings of the design, automation and test in Europe conference and exhibition, pp 775–780
- Falk J, Zebelein C, Haubelt C, Teich J (2011) A rule-based static dataflow clustering algorithm for efficient embedded software synthesis. In: Proceedings of the design, automation and test in Europe conference and exhibition
- Frigo M, Johnson SG (2005) The design and implementation of FFTW3. Proc IEEE 93(2):216–231
- Ha S, Kim S, Lee C, Yi Y, Kwon S, Joo YP (2007) PeaCE: a hardware–software codesign environment for multimedia embedded systems. ACM Trans Des Autom Electron Syst 12, Article no. 24
- Heulot J, Pelcat M, Nezan J, Oliva Y, Aridhi S, Bhattacharyya SS (2014) Just-in-time scheduling techniques for multicore signal processing systems. In: Proceedings of the IEEE global conference on signal and information processing, Atlanta, Georgia, pp 175–179
- Hsu C, Ko M, Bhattacharyya SS (2005) Software synthesis from the dataflow interchange format. In: Proceedings of the international workshop on software and compilers for embedded systems, Dallas, Texas, pp 37–49
- Johnston WM, Hanna J, Millar RJ (2004) Advances in dataflow programming languages. ACM Comput Surv 36(1):1–34
- Jones MM, Mestre J, Scholz B (2017) Towards memory-optimal schedules for SDF. In: Proceedings of the European conference on parallel processing, pp 94–105
- Keutzer K, Newton AR, Rabaey JM, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE Trans Comput-Aided Des Integr Circuits Syst 19(12):1523–1543
- Ko M, Zissulescu C, Puthenpurayil S, Bhattacharyya SS, Kienhuis B, Deprettere E (2007) Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation. IEEE Trans Signal Process 55(6):3126–3138
- Lee EA (2006) The problem with threads. Computer 39(5):33–42
- Lee EA, Ha S (1989) Scheduling strategies for multiprocessor real time DSP. Proc Glob Telecommun Conf 2:1279–1283
- Lee EA, Messerschmitt DG (1987) Synchronous dataflow. Proc IEEE 75(9):1235–1245
- Lee EA, Parks TM (1995) Dataflow process networks. Proc IEEE 83(5):773–799
- Lee K (2018) Design optimization of embedded signal processing systems for target detection. Ph.D. thesis. Department of Electrical and Computer Engineering, University of Maryland, College Park
- Lee K, Riggan BS, Bhattacharyya SS (2017) An accumulative fusion architecture for discriminating people and vehicles using acoustic and seismic signals. In: Proceedings of the international conference on acoustics, speech, and signal processing, New Orleans, Louisiana, pp 2976–2980
- Lee K, Riggan BS, Bhattacharyya SS (2017) An optimized embedded target detection system using acoustic and seismic sensors.

- In: Proceedings of the European signal processing conference, Kos Island, Greece, pp 1021–1025
27. Lin S, Liu Y, Lee K, Li L, Plishker W, Bhattacharyya SS (2017) The DSPCAD framework for modeling and synthesis of signal processing systems. In: Ha S, Teich J (eds) Handbook of hardware/software codesign. Springer, Berlin, pp 1–35
 28. Liu W, Gu Z, Xu J (2009) Efficient software synthesis for dynamic single appearance scheduling of synchronous dataflow. *IEEE Embed Syst Lett* 1(3):69–72
 29. Ma M, Sakellariou R (2018) Reducing code size in scheduling synchronous dataflow graphs on multicore systems. In: Proceedings of the workshops on parallel programming and runtime management techniques for manycore architectures and design tools and architectures for multicore embedded computing platforms, pp 57–62
 30. Nabritt SM, Damarla T, Chatters G (2015) Personnel and vehicle data collection at Aberdeen proving ground (APG) and its distribution for research. Technical report ARL-MR-0909. US Army Research Laboratory
 31. Neuendorffer S (2004) Actor-oriented metaprogramming. Ph.D. thesis. Department of Electrical Engineering and Computer Sciences, University of California at Berkeley
 32. Oh H, Ha S (2003) Memory-optimized software synthesis from dataflow program graphs with large size data samples. *EURASIP J Adv Signal Process* 2003:650391
 33. Pelcat M, Piat J, Wipliez M, Aridhi S, Nezan JF (2009) An open framework for rapid prototyping of signal processing applications. *EURASIP J Embed Syst* 2009, Article no. 11
 34. Pino JL, Ha S, Lee EA, Buck JT (1995) Software synthesis for DSP using Ptolemy. *J VLSI Signal Process* 9:7–21
 35. Plishker W, Sane N, Kiemb M, Anand K, Bhattacharyya SS (2008) Functional DIF for rapid prototyping. In: Proceedings of the international symposium on rapid system prototyping, Monterey, California, pp 17–23
 36. Plishker W, Sane N, Kiemb M, Bhattacharyya SS (2008) Heterogeneous design in functional DIF. In: Proceedings of the international workshop on systems, architectures, modeling, and simulation, Samos, Greece, pp 157–166
 37. Putnam LH, Myers W (2003) Five core metrics: the intelligence behind successful software management. Dorset House, Oxford
 38. Ragan-Kelley J, Adams A, Paris S, Levoy MS, Amarasinghe S, Durand F (2012) Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans Graph* 31(4):1–12
 39. Raina A (2017) Synthesis of embedded software using dataflow schedule graphs. Master's thesis. Department of Electrical and Computer Engineering, University of Maryland, College Park
 40. Roquier G, Bezati E, Mattavelli M (2012) Hardware and software synthesis of heterogeneous systems from dataflow programs. *J Electr Comput Eng* 2012:1–11
 41. Shen C, Plishker W, Wu H, Bhattacharyya SS (2010) A lightweight dataflow approach for design and implementation of SDR systems. In: Proceedings of the wireless innovation conference and product exposition, Washington DC, USA, pp 640–645
 42. Shen C, Wu S, Sane N, Wu H, Plishker W, Bhattacharyya SS (2012) Design and synthesis for multimedia systems using the targeted dataflow interchange format. *IEEE Trans Multimed* 14(3):630–640
 43. Silva J, Sklyarov V, Skliarova I (2015) Comparison of on-chip communications in Zynq-7000 all programmable systems-on-chip. *IEEE Embed Syst Lett* 7(1):31–34
 44. Sriram S, Bhattacharyya SS (2009) Embedded multiprocessors: scheduling and synchronization, 2nd edn. CRC Press, Boca Raton
 45. Sung W, Kim J, Ha S (1998) Memory efficient software synthesis from dataflow graph. In: Proceedings of the international symposium on system synthesis, pp 137–142
 46. Tendulkar P (2014) Mapping and scheduling on multi-core processors using SMT solvers. Technical report. Universite de Grenoble I—Joseph Fourier
 47. Wu H, Shen C, Sane N, Plishker W, Bhattacharyya SS (2011) A model-based schedule representation for heterogeneous mapping of dataflow graphs. In: Proceedings of the international heterogeneity in computing workshop, Anchorage, Alaska, pp 66–77
 48. Yviquel H, Sanchez A, Jaaskelainen P, Takala J, Raulet M, Casseau E (2014) Efficient software synthesis of dynamic dataflow programs. In: Proceedings of the international conference on acoustics, speech, and signal processing, pp 4988–4992
 49. Zebelein C, Haubelt C, Falk J, Teich J (2013) Model-based representation of schedules for dataflow graphs. In: Proceedings of workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, pp 105–116

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.