SDF^3: SDF For Free*

Sander Stuijk, Marc Geilen and Twan Basten
Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands.
{s.stuijk,m.c.w.geilen,a.a.basten}@tue.nl

Abstract. SDF^3 is a tool for generating random Synchronous DataFlow Graphs (SDFGs), if desirable with certain guaranteed properties like strongly connectedness. It includes an extensive library of SDFG analysis and transformation algorithms as well as functionality to visualize them. The tool can create SDFG benchmarks that mimic DSP or multimedia applications.

1. Introduction

DSP and multimedia applications are applications in which a set of operations or cooperating tasks is executed periodically. These applications are often mapped to parallel platforms which enables parallel execution and pipelining of the operations or the tasks. Often, cyclic data dependencies exist spanning operation or task execution in different periods. Synchronous DataFlow Graphs (SDFGs) [6] are suitable for modeling both parallel and pipelined processing and cyclic dependencies. An important reason for the popularity of SDFGs is the existence of analysis techniques. Development of new SDF-based design techniques is hampered however by the availability of only a limited set of test graphs. Inspired by Task Graphs For Free [2] that can only generate acyclic task graphs with non-pipelined point-to-point communication, we developed a tool called *SDF For Free (SDF^3)*. It generates random sets of SDFGs, with support to analyze and visualize SDFGs. The tool with C++ source code is freely available from `http://www.es.ele.tue.nl/sdf3`. It allows researchers and designers to generate sets of SDFGs for evaluation purposes, and to easily compare new analysis techniques to existing ones.

2. Synchronous Dataflow Graphs

An example of an SDFG is depicted in Fig. 1. The nodes of an SDFG are called *actors*; they represent functions that are computed by reading *tokens* (data items) from their input ports, and writing the results of the computation as tokens on the output ports. An essential property of SDFGs is that every time an actor *fires* (performs a computation) it consumes the same amount of tokens from its input ports and produces the same amount of tokens on its output ports. These amounts are called the port *rates* and are visualized at both ends of the edges. The edges in the graph, called *channels*, represent data that is communicated from one actor to another. A channel may contain initial tokens. In Fig. 1, the number of initial tokens is attached to the channel label. For analysis purposes one typically abstracts from the actual functions which actors compute and the values of tokens.

Consistency and absence of deadlock are two important properties for SDFGs [1, 6]. Consistency is determined by the port rates. Any SDFG which is not consistent requires unbounded memory to execute or deadlocks. When an SDFG deadlocks, no actor is able to fire, which is either due to inconsistency or due to an insufficient number of tokens in a cycle of the graph.

3. SDFG Generation

Our SDFG generation algorithm constructs graphs which are connected, consistent, and deadlock-free. Unconnected SDFGs can always be constructed by combining two or more SDFGs generated with the algorithm.

A user specifies in a configuration file the most important parameters determining the characteristics of the graph. These parameters are the fixed number of actors in the graph and the average and variance of their degree (i.e. the number of input and output ports) and port rates. The actual degree of actors and the port rates are random values picked by a random number generator [7]. To get a better control over the characteristics of the generated graphs, a minimum and maximum value, bounding the range of possible values, can also be specified.

Algorithm 1 shows pseudo-code for the SDFG generation algorithm. It starts by constructing a connected SDFG containing the user-specified number of actors (line 2-14). All ports are assigned a random rate which can make the SDFG inconsistent. At line 16, consistency is checked using the algorithm described in [1]. Whenever an inconsistency is found, the rates of the involved ports are changed such that the SDFG becomes consistent. Next, the procedure DISTRIBUTETOKENS distributes tokens over the channels in the graph making it deadlock free. This is done by selecting a random channel in the graph and adding a token to it and computing the throughput of the graph. If the graph is not deadlock free, the throughput will be zero and more tokens need to be added. This process is repeated till the graph becomes deadlock free. At this point, the procedure DISTRIBUTETOKENS may continue adding a random

*This work was supported by the Dutch Science Foundation NWO, project 612.064.206, PROMES, and the EU, project IST-004042, Betsy.

Algorithm 1 Generate random SDF graph

Input: Number of actors ($nrActors$) and specification of the properties ($propSpec$) of the SDFG.
Result: An SDFG (A, C).

1: **procedure** RANDOMSDFG($nrActors$,$propSpec$)
2: Create actor a with random (according to $propSpec$) input/output ports InP_a / $OutP_a$
3: $A \leftarrow \{a\}$; $C \leftarrow \emptyset$
4: $OutP \leftarrow OutP_a$; $InP \leftarrow InP_a$
5: **while** $|A| < nrActors$ **do**
6: Create actor a with random ports InP_a / $OutP_a$
7: $A \leftarrow A \bigcup \{a\}$
8: Connect actor to graph via random channel (p_o, p_i)
9: $OutP \leftarrow (OutP \bigcup OutP_a) \backslash \{p_o\}$
10: $InP \leftarrow (InP \bigcup InP_a) \backslash \{p_i\}$
11: **while** $InP \neq \emptyset$ and $OutP \neq \emptyset$ **do**
12: Get random ports $p_o \in OutP$ and $p_i \in InP$
13: $C \leftarrow C \bigcup \{(p_o, p_i)\}$
14: $OutP \leftarrow OutP \backslash \{p_o\}$; $InP \leftarrow InP \backslash \{p_i\}$
15: Remove all ports $InP \bigcup OutP$ from actors
16: MAKECONSISTENT(A,C)
17: DISTRIBUTETOKENS(A,C)
18: ANNOTATE(A,C)

number of additional tokens to a set of randomly selected channels.

For many SDFG analysis algorithms, properties must be assigned to the actors, tokens, and channels, or the SDFG as a whole. For example, algorithms which deal with the throughput of a graph require timing annotations on actors. Buffer sizing algorithms typically require token sizes which can be annotated to the channels. SDF^3 contains functions to assign randomly selected values to actors and channels which represent these properties. Currently, it can assign an execution time and memory requirement to each actor, token sizes and buffer sizes to channels, and set a throughput constraint on the graph. This annotation mechanism can easily be extended by users of the tool.

By default, the SDFGs generated by Algorithm 1 are connected in an arbitrary way. If desired, the tool can restrict the connections between the actors making the graph a chain, a-cyclic, or strongly connected. DSP and multimedia applications, which are often of these forms, can easily be mimicked in this way.

4. SDFG Analysis and Visualization

SDF^3 implements, besides generation of random graphs, a library offering the following SDFG analysis and transformation algorithms.

- (Repetition vector) It can compute the repetition vector of an SDFG. The repetition vector gives the number of times each actor should be fired in order to bring the SDFG back to its original state. It is, for example, used to check the consistency of an SDFG.
- (Convert SDFG to corresponding HSDFG) It implements the algorithm from [10] to convert any arbitrary SDFG into its corresponding homogeneous SDFG (HSDFG). HSDFGs are a special class of SDFGs with all port rates equal to 1, implying that all repetition vector entries are 1. HSDFGs are used in many scheduling and throughput analysis algorithms.
- (Unfold HSDFG) It can perform unfolding of an HSDFG. Unfolding, i.e. replication of actors in an HSDFG, is needed to create additional freedom for HSDFG scheduling algorithms, potentially allowing them to generate a schedule with a higher throughput.
- (Scheduling) It contains a list-scheduler, which is the basic SDFG scheduling technique for multi-processor systems and a single appearance scheduler, typically used for DSP applications, which minimizes code size.
- (MCM analysis) It implements Karp's, Howard's, and Young-Tarjan-Orlin's MCM algorithms. These algorithms can be used to compute the maximally obtainable throughput of an SDFG after converting it to its corresponding HSDFG.
- (Self-timed execution) It can perform a self-timed execution which can be used as an alternative means to compute the throughput of an SDFG. This technique is explained in [5], where SDF^3 is used to generate a benchmark of SDFGs and to compare the self-timed execution method for computing throughput to techniques which use MCM analysis.
- (Buffer sizing) It contains algorithms to compute the minimal buffer sizes [4] and the trade-offs between buffer sizes and the throughput of an SDFG [11]. Buffer sizes are modeled into an SDFG. Self-timed execution is used to compute throughput. Dependencies of actor firings on each other and cycles of these dependencies, which limit throughput, are also found. By increasing the buffer size of selected channels on these cycles throughput can be increased.
- (XML-based format for SDFGs) It supports an XML-based format for SDFGs which enables simple exchange of graphs between different tools.
- (Visualization) It offers a function to visualize SDFGs through the popular graph visualization tool dotty [3].

5. DSP Synthesis

Both hardware and software synthesis of DSP applications modeled as SDFGs has been studied extensively [8]. Synthesis involves mapping an SDFG to a target platform and ordering the actor executions. Typically, single appearance schedulers are used to minimize the code size when an application is mapped to a single processor system. A list-scheduler is often used to map an SDFG to a parallel platform. This scheduler requires that an SDFG is converted into its corresponding HSDFG. Furthermore, unfolding of

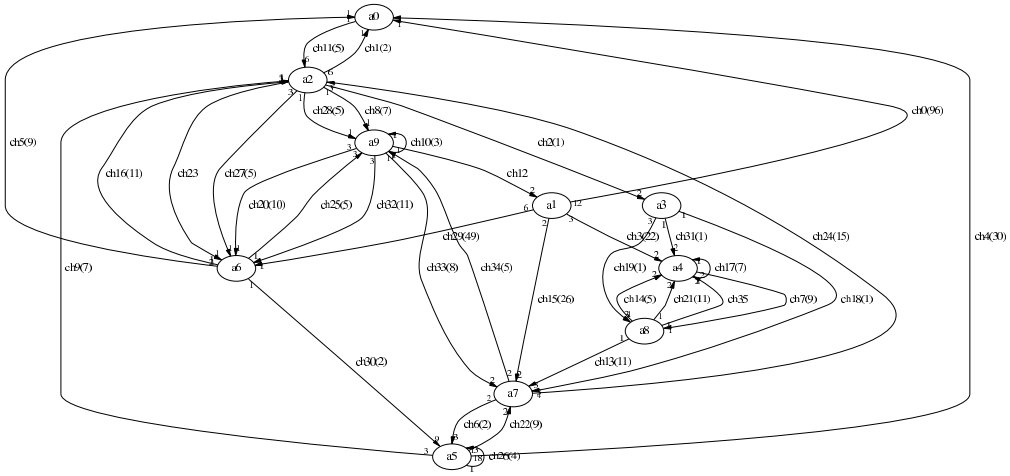

Figure 1. An SDFG generated and visualized with SDF^3.

the HSDFG is often used to increase the scheduling freedom. MCM algorithms are used to analyze the throughput of an HSDFG model of a (mapped) DSP application. All these basic DSP synthesis algorithms are available in the SDF^3 library.

6. Multiprocessor Design Flow

SDFGs are often used to model multimedia applications which have to be mapped to a multiprocessor system. The idea is that design decisions made during the design flow are modeled into the SDFGs. Following [9], a design flow for mapping an application modeled as an SDFG onto a multiprocessor system with a predictable timing behavior may look as follows. First, the application graph must be checked for consistency and absence of deadlock. Next, actors are mapped to the various processors in the system. When multiple actors are assigned to the same processor, a schedule ordering their firings is constructed. A list-scheduler or single appearance scheduler can be used for this. Next, a decision is made on the storage space assigned to each of the channels in the SDFG. To do this, the designer must make a trade-off between the used storage space and the throughput of the application. This storage space must also be allocated in the various memories available in the system. Finally, the channels in the SDFG must be mapped to the communication resources and token communications must be scheduled on them. The SDF^3 library offers many basic functions which are needed in the various steps of the design flow. This makes it a very useful instrument for setting-up an SDFG-based flow.

7. Conclusion

SDF^3 is a tool for generating SDFGs, a popular model of DSP and multimedia computation. The user can control the characteristics of the graphs by specifying bounds, averages and variances on various aspects of an SDFG. The tool implements a library offering many SDF analysis and transformation techniques as well as a function to visualize SDFGs. It is very simple to extend the graph generation tool and integrate other techniques on top of the library.

References

[1] S. Bhattacharyya, P. Murthy, and E. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[2] R. Dick, D. Rhodes, and W. Wolf. TGFF: task graphs for free. In *CODES*, pages 97–101. IEEE, 1998.

[3] E. Gansner and S. North. An open graph visualization system and its applications to software engineering. *Software: practice and experience*, 30(11):1203–1233, 2000.

[4] M. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model-checking. In *DAC'05, Proc.*, pages 819–824. IEEE, 2005.

[5] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi. Throughput analysis of synchronous data flow graphs. In *ACSD'06, Proc.* IEEE, 2006.

[6] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[7] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[8] J. Pino, S. Ha, E. Lee, and J. Buck. Software synthesis for DSP using Ptolemy. *Journal of VLSI Signal Processing*, 9(1):7–21, 1995.

[9] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *CASES'03, Proc.*, pages 63–72. ACM, 2003.

[10] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors Scheduling and Synchronization*. Marcel Dekker, 2000.

[11] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. *To appear in DAC'06, Proc.* ACM, 2006.