Efficient Code Synthesis from Synchronous Dataflow Graphs

Dag Björklund
Turku Centre for Computer Science (TUCS)
Åbo Akademi University
Lemminkisenkatu 14 A, FIN-20520
Turku, Finland
dbjorklu@abo.fi

Abstract

We present a novel approach for efficient code synthesis from Synchronous Dataflow specifications. The method avoids duplication of code blocks when compiling SDF graphs regardless of whether a single appearance schedule can be found for the graph or not. This also means that we can use schedules that require minimal buffer memory, but are not single appearance schedules. The method has been developed within the compiler for the Rialto language, which we have developed for use as an intermediate language for code synthesis from heterogeneous models of computation. The optimization technique presented in the paper can, however, very well also be used without the Rialto language.

1. Introduction

Dataflow is a natural paradigm for describing DSP applications. Synchronous dataflow (SDF), originally developed by Karp and Miller [8] and by Lee and Messerschmitt [9], is a special case of dataflow, where a static schedule for the system can be computed at compile time, since the number of data samples produced or consumed by each node on each invocation is specified beforehand. This reduces the run-time overhead usually associated with dataflow. A simple SDF graph is shown in Figure 1. The numbers on the edges specify how many tokens the nodes consume and produce each time they are invoked. Node A produces two tokens and B consumes one, which means that B has to be executed twice for each invocation of A. A valid schedule for this graph is for example $ABCCBCC$. The schedule can also be written as $A(2B(2C))$ to highlight the repetitive invocation pattern using *schedule loops*. The schedule loops are a shorthand for writing the schedules, but can also exploited by code synthesis tools for achieving efficient code. This schedule is also a *single appearance schedule*, since each lexical element appears only once. Single appearance schedules do not exist for all SDF graphs.

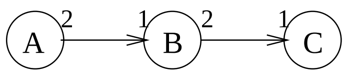

$$\begin{tikzcd} A \arrow[r, "2\quad 1"] & B \arrow[r, "2\quad 1"] & C \end{tikzcd}$$

Figure 1. A Simple SDF graph

[0]0-7803-8509-8/04/$20.00 ©2004 IEEE

1.1. Code Synthesis from SDF Graphs

Quite efficient model compilers can be developed for SDF graphs, that exploit the restrictions the SDF model of computation enforces on a design, e.g. [3, 2]. It is desirable to generate inline code for the actors in a graph in order to avoid function call overhead, which can be significant since the actors can be simple operations. This can, however, lead to an explosion of the code size, which is intolerable in embedded systems. The traditional techniques for minimising program memory requirement, while using inline code are based on finding single appearance schedules for the SDF graphs.

If we were to do a simple straight code generation using the schedule $ABCCBCC$ for the example, we would get undesirable code duplication as below:

```
code block for A;
code block for B;
code block for C;
code block for C;
code block for B;
code block for C;
code block for C;
```

The schedule loops in a single appearance schedule can be translated into loops in the generated code, which avoids duplication of the inlined code blocks. Using the single appearance schedule $A(2B(2C))$ we can generate the code below, which contains each actor code block only once.

```
code block for A;
for(i=0;i<2;i++) {
   code block for B;
   for(j=0;j<2;j++) {
     code block for C;
   }
}
```

Another consideration when attempting to generate memory efficient code from SDF graphs is the *buffer memory requirement*, that is the maximum number of tokens stored in the buffers, or edges, during an execution of a schedule. In the traditional approaches there is a tradeoff between buffer memory requirement and code size optimisation. Finding a single appearance schedule to minimize code size, may result in large buffer memory requirements.

1.2. The Rialto Approach

In this paper we present a different approach for synthesis of efficient target language code from SDF specifications and also other models of computation. The method has been developed within the compiler of a language called Rialto. The Rialto language is a simple textual language with formal semantics designed for modelling systems in multiple models of computation [4], or for use as an intermediate language for code synthesis from models in different visual languages. It was originally designed for representing UML statecharts [5] and has been used for combining different behavioral UML models [6]. The translation from SDF or other graphical models into Rialto is simple, and can be performed by a tool. We are for example using the System Modeling Workbench (SMW) [13] to experiment on model translations. The SMW is based on metamodeling, which allows new modeling paradigms to be easily added. The Rialto language can thus be made transparent to the user when using it as a intermediate language for code synthesis.

Rialto code can be compiled into target language, by first translating it into finite state machines (FSMs), which can be reduced using S-GRAPHS, introduced in the POLIS approach [1]. The FSMs are constructed from the structural operational semantics of Rialto. In the case of SDF, the translation into FSM actually creates a machine where each invocation of a node in a schedule becomes a state, and the node invocations become unguarded transitions between the states, with the executions of their code blocks as side effects.

The optimization technique is then a matter of reducing this state space, and the redundancies in the machine as much as possible using the S-GRAPHS. An SDF static schedule could also very easily be translated directly into an S-GRAPH, instead of using Rialto as an intermediate step. The benefit of Rialto comes when using other models of computation in combination with SDF graphs.

What makes this approach interesting, is that the S-GRAPH technique removes duplication of SDF node code blocks regardless of the static schedule computed for the graph. It does not use schedule loops and thus we do not need to find a single appearance schedule as in the traditional approaches. This means that we can focus on buffer memory requirement minimisation in the scheduling and get the small code size for free, avoiding the tradeoff between the two. The penalty of our method, is a larger runtime overhead.

2. The Rialto Language and Optimizing Compiler

The SDF compilation approach presented in this paper has been developed within the compiler of the Rialto language; however, it can be used just as well without using Rialto as an intermediate format. Rialto is developed for capturing heterogeneous designs, and the benefits of using it arises when doing code synthesis from such models, as we will briefly explain in Section 6. The semantics of Rialto has been divided into two levels. The first defines the behavior of each statement in the language that represent concepts like concurrency, state, interrupts etc. common to several models of computation. The statements leave nondeterminism in the scheduling of parallel actors, which is resolved by the second level of semantics, called *scheduling policies* that represent different computational models, like synchronous dataflow.

Figure 2 illustrates the use of Rialto as an intermediate language for code synthesis from different modeling languages. From the Rialto code, we construct FSMs using the operational semantics. Then we apply a set of optimization steps before proceeding to target code generation. In this section we will, as shortly as we can, introduce the Rialto language and how it is compiled, in terms of an SDF graph example.

Figure 3 shows a Rialto translation of the simple SDF graph from the previous example. The `state` blocks are the key abstraction in the language. They can be hierarchical and parallel and they can represent different actors in different models of computation e.g. states in a statechart, parallel threads or as in the figure, nodes in an SDF graph. The `state` blocks are labeled according to the names in the SDF graph. The `par` statement in line 4 in the example, activates the labels `A,B` and `C` in parallel, where after there is nondeterminism in which one to execute. This choice is made by the scheduling policy in the current `state` block. In this case the SDF policy (line 2).

Each `state` block in a Rialto program can be assigned a different scheduling policy to reflect different computational models. In the example, the `upsampler` block is scheduled by a SDF policy, which is given the static precomputed schedule `A,B,C,C,B,C,C`. The schedules are given to the policy without schedule loops, as they are not

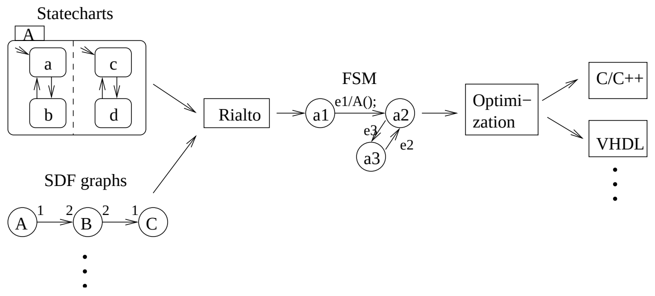


Figure 2. The Rialto Compilation Process

```
upsampler: state
  policy SDF(A,B,C,C,B,C,C);
begin
  par(A,B,C)
  A: state
    # code block for A
  end
  B: state
    # code block for B
  end
  C: state
    # code block for C
  end
end
```

Figure 3. The Rialto translation of the SDF graph in Figure 1

needed in our approach. The schedule can for example be computed using a very simple heuristic algorithm that constructs a minimal buffer size schedule as found in [2].

A program is executed by repeatedly running the policy of the topmost state in the hierarchy. The topmost policy will then schedule the blocks down in the hierarchy, which can have different policies assigned to them. The entity that calls the scheduling policy of the top-level state can be thought of as a global clock in the system.

The state of a program σ is represented by the set of active labels α, which represents the control state, and the data state $\mathcal{E}$, that is $\sigma = \langle \alpha, \mathcal{E} \rangle$.

A scheduling policy function accepts as parameters the current state of the program σ and the set of blocked labels β and returns the next state of the program. In addition, the SDF policy is given a schedule S. The schedule is a sequence of labels of `state` blocks. For example $S = [A, B, C, C, B, C, C]$ in the previous example. The SDF scheduling algorithm simply executes one period of a schedule and returns the state of the system after execution:

```
RUN($\sigma, \beta, S$)
1   for each $s$ in $S$ do
2       $\sigma \leftarrow s.RUN(\sigma, \beta, S)$
3   return $\sigma'$
```

$$\frac{policy(l) = SDF \quad l \in \sigma.\alpha \quad \langle \sigma, \emptyset, s \rangle \overset{S[s].RUN}{\rightsquigarrow} \langle \sigma', \emptyset, s \rangle}{\langle \sigma, \beta, s \rangle \rightarrow \langle \sigma', \beta, s+1 \rangle}$$

$$\frac{policy(l) = SDF \quad s = end}{\langle \sigma, \beta, s \rangle \rightarrow \langle \sigma, \Lambda, 0 \rangle}$$

Figure 4. An SDF scheduling algorithm a) and a formalisation b)

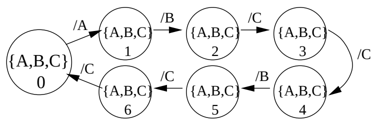


Figure 5. Computation as an FSM

Calling $l.RUN(\sigma, \beta)$ for a label l belonging to a `state`, executes the policy attached to it.

We can develop different SDF policies for Rialto depending on the level of atomicity we want. This is mainly interesting for cases were we use another computational model to manage SDF computation, and we wish to choose the points at which the SDF computation can be interrupted by the higher level model. The SDF algorithm presented, executes a whole schedule period without allowing interrupts by other actors.

In order to construct an FSM from a Rialto program, we formalize the scheduling algorithms using structural operation rules. The statements are also defined the same way, but we need not go into that here. The SDF policy is defined by the operational rules shown in Figure 4.

Here we iterate the schedule S by indexing it with index variable s. So $S[0]$ returns the first item in the schedule sequence. The state σ on which the statements operate can be called *micro state*, while the scheduling policies operate on a *macro state* represented by the tuple $\langle \sigma, \beta, s \rangle$, where s is the SDF indexing variable. The first rule executes a block with label l whose policy is SDF and belongs to the active set of of σ ($l \in \sigma.\alpha$), by executing the next actor in its SDF schedule. The state is updated to σ', the blocked set β is left unchanged and the variable s is increased by one. The second rule is chosen when the we have reached the end of the schedule. The termination of the step is indicated by the Λ, and the s variable is reset to 0.

Using these rules, we can construct an FSM, where the states are the macros states $\langle \sigma, \beta, s \rangle$ and the transitions are calculated from the operational rules. From the example program we obtain the FSM in Figure 5. The states are labeled with the active set, which remains unchanged, and the s variable the steps through the schedule. The transitions are unguarded and update the data state by executing the code blocks for the nodes as side effects. We notice that this is nothing but a sequential computation of the schedule. In a combination with statecharts or other computational models, there could be guarded transitions to other states etc.

3. Optimized Code Synthesis

The FSM representation constructed from the Rialto operational rules can be translated into an S-GRAPH, which can be significantly reduced. The S-GRAPH technique was introduced in the Polis approach, and we have added several additional reduction steps on the graphs. Although we will here explain how FSMs are translated into S-GRAPHS, we could also translate an SDF static schedule directly. After all, the FSMs achieved from the SDF graphs as explained in the previous section, are merely sequences of unguarded transitions, that actually form the static schedule invocation sequence.

An S-GRAPH is a directed acyclic graph used to describe a decision tree with assignments. An S-GRAPH consists of a set of vertices, which contains four types of vertices: BEGIN, END, TEST and ASSIGN. Every S-GRAPH has one vertex of type BEGIN, called the source and one vertex of type END, called the sink. All other vertices are of type TEST or ASSIGN. Each TEST vertex v has two children, which are called `true(v)` and `false(v)`. Each BEGIN or ASSIGN vertex u has only one child `next(u)`. Each vertex is labeled with a function. Two nodes are *isomorphic* if they have the same label, and their child or children are isomorphic. If there are no two isomorphic nodes in an S-GRAPH, it is said to be *reduced*.

An FSM can be translated into an S-GRAPH by using TEST nodes to check the current state and events and guards, and ASSIGN nodes to do state transitions and execute side effects. In our SDF example we only have unguarded transitions. The FSM from Figure 5 can be translated into the S-GRAPH shown in Figure 6 a). The dashed lines denote false-branches; solid lines denote true-branches. We use a variable S to record the state. The node labeled S=ABC0 for example is a TEST node checking if the machine is in state ABC0, the A node is an ASSIGN node that executes the code block for node A.

The micro states in the FSM differ only in the s indexing variable that gives the position in the schedule. So these micro transitions can be translated into a simple increment of the state variable, which is easily recognised by the Rialto compiler as well. On traversal of the S-GRAPH, finds the current state, executes a side effect and does one state transition, in this case by incrementing it by one.

We see obvious possibilities for combining isomorphic nodes in the graph. For example nodes 14 and 15 are isomorphic since they have the same label C, and their children are isomorphic having the label S++. By combining all isomorphic nodes, we get the reduced graph in Figure 6 b), where the code blocks A,B and C now appear only once, as well as the increment of the S variable.

We have developed the S-GRAPH reduction further for our code generation purposes. We can combine TEST nodes that share the same `true` branch, by `or`:ing the guards together. The result of this operation on the example is shown in Figure 6 c). This reduces the nodes a lot and makes the code generation easier and more efficient. It also presents us with opportunities to find repetition patterns in the schedule. In the traditional approaches, repetitions, or schedule loops are found at scheduling time, while here some repetitions can be found by the Rialto compiler, with the help of S-GRAPHs.

Since we now have all occurrences of a node in the schedule gathered in one TEST node, we can look for repetitions inside the node, or with its neighbours. In the example, we see that in the `or` node $S = ABC2|ABC3|ABC5|ABC6$, the C node is always executed twice in succession. We can thus reduce the `or` node into $S = ABC2|ABC5$ as in Figure 6 d). The C node is then executed twice each time the guard of the `or` node evaluates to true. This is indicated by the label $2 * C$ on the ASSIGN node 11.

This reveals still further reduction opportunities. We find that in node 2 and 3 of the new graph, each element in the `or` list in node 3, is a successor of the corresponding element in node 2. This means that C is always executed two times after an execution of B. We can thus combine these two nodes as done in Figure 6 e). During this process, we have achieved a desirable reduction of the states by the removal of the states $ABC2$, $ABC3$, $ABC5$ and $ABC6$, that are not needed anymore.

3.1. Target Code Generation

From the optimized S-GRAPH it is now an easy task to generate target code in different languages. The easiest way of translating the S-GRAPH into target language is to use `goto` statements in C for example. The C code below is a direct translation of the graph in Figure 6 e).

```
enum State {ABC0=0,ABC1=1,ABC4=2} S;
while(true) {
  l1: if (S==ABC0)
    goto l9;
  else
    goto l2;
  l2: if (S==ABC1 || S==ABC4)
    goto l10;
  else
    goto l11;
  l9: // code block for A
  goto l11;
  l10: // code block for B
  for ( i=0; 2; i++ )
```

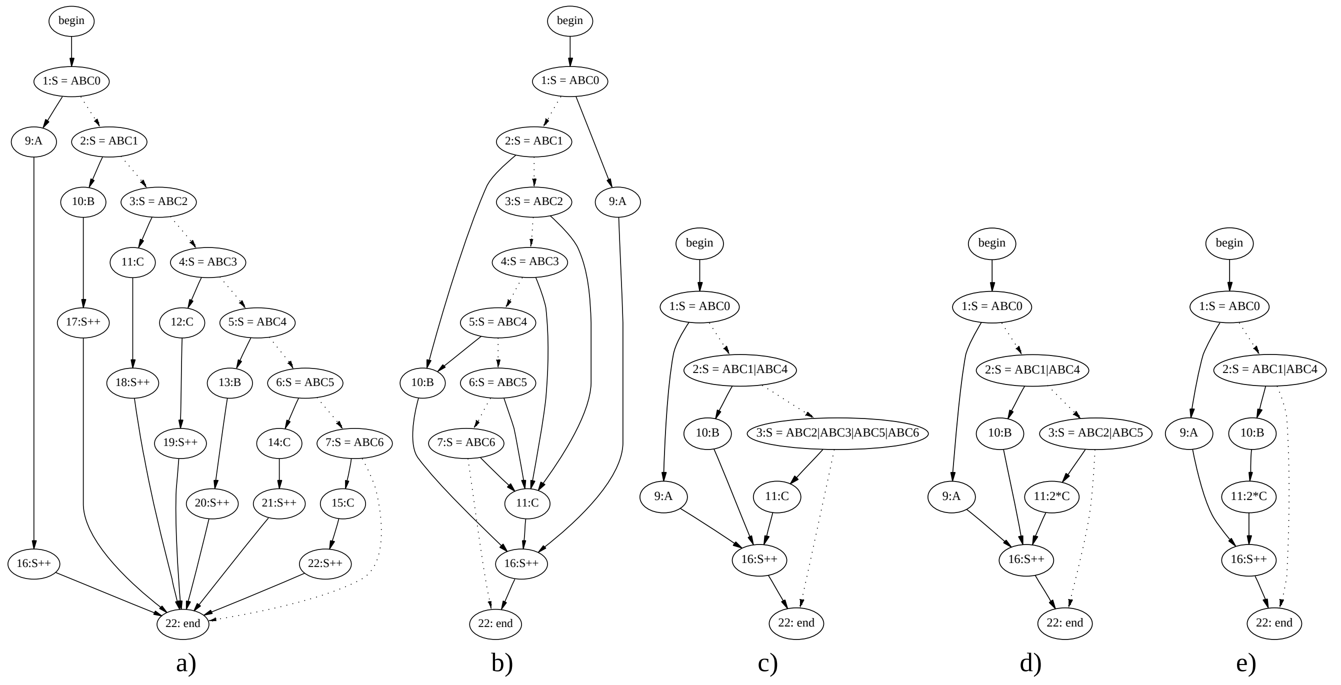


Figure 6. Four reduction steps for S-Graphs

```
    // code block for C
  l11: S++%4;
}
```

The variable S representing the state is of the type enumerating the state-space. The label `l1` corresponds to node 1 in the graph etc. and the `goto` statements perform jumps to the labels according to guard evaluations from the TEST nodes. The double execution of the `C` node is performed by the `for` loop. The whole thing is encapsulated by an infinite loop, and the `S++` increment is exchanged to `S++%4`, which makes the code run periodically. Notice that the code blocks `A,B` and `C` appear only once, although we did not use a single appearance schedule.

Using this approach for compilation of larger models results in numerous comparison operations in the `or` guards at each iteration. We can reduce this overhead dramatically by representing the state `S` by a bit-vector, where a '1' bit in position i means the i^{th} state is active. The `or` states can then be combined at compile time using the C bitwise or `|`. This reduces the guard evaluations to a single bitwise and `&`, to see if we are in one of many states. We get to the next state by doing a shift operation on `S` i.e. `S<<=1`. For larger models, we have to divide the state space into several integer variables, since a 32 bit integer for example, can now only represent 32 states. We can do this with very little extra overhead however. Also these state variables will require some extra memory, but it should not be too significant. Using this technique, we can rewrite the previous code as below:

```
enum State {ABC0=1,ABC1=2,ABC4=4} S;
for ( S=ABC0; S <= ABC4; S<<=1 ) {
  if (S & ABC0)
    // code block for A;
  else if (S & (ABC1|ABC4) {
    // code block for B;
    for ( i=0; i<2; i++ )
      // code block for C;
}
```

We have here also used an alternative bottom-up approach for translating the S-Graph, which produces a set of `if-else if` statements instead of the `goto` statements. We have also used a `for` loop inplace of the `while` loop. The loop computes one period of the schedule. This approach produces much more compact code, but is not necessarily any more efficient. We could apparently still reduce the code in this case by factoring out the execution of `A`, since it is run only once at the beginning, and removing the `S&(ABC1|ABC4)` guard.

4. Buffer Memory versus Code Size

In traditional code synthesis approaches from synchronous dataflow specifications, there is a trade-off between code size and buffer memory size. There is a choice between first attempting to create a schedule that optimizes the code size, where-after techniques for minimising the buffer memory can be applied, or first developing a schedule that minimises the buffer memory and then trying to reduce the code size as much as possible. The code size

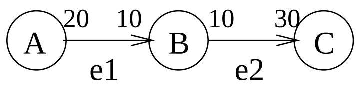


Figure 7. An SDF graph that does not have a BMLB schedule

	A	B	B	A	B	C	B	A	B	B	C
e1	20	10	0	20	10	10	0	20	10	0	0
e2	0	10	20	20	30	0	10	10	20	30	0

Table 1. A minimal buffer memory schedule for the SDF in Figure 7

minimisation is performed by searching for a single appearance schedule, which can be used to create loop structures with no code duplication.

The thing that makes the Rialto S-GRAPH reduction for SDF graphs just presented interesting, is that it always results in implementations without code duplication, regardless of whether a single appearance schedule can be found or not, or whether first emphasis is put on code size reduction or buffer memory reduction. In other words we can put all emphasis on the buffer memory optimisation, while optimal code size is received for free.

The traditional techniques are optimal when a single appearance schedule whose buffer memory requirement equals the *Buffer memory lower bound* (BMLB) of the SDF graph. The BMLB of a graph is the minimal amount of memory required for the buffers or edges of the graph. A BMLB schedule is a single appearance schedule that satisfies the buffer memory lower bound. The single appearance schedule A(2B(2C)) we used for the example graph results in minimal buffer memories.

Figure 7 shows an example where a BMLB schedule cannot be found. The two edges are labeled $e1$ and $e2$. The BMLB for this graph is 50 (see e.g. [3] for an explanation of the calculation). We can easily find a single appearance schedule for the graph, for example (3A)(6B)(2C), but it has a buffer memory requirement of 120. In other words, giving first priority to code size reduction gives us a buffer memory overhead of 240%.

We also find that we can easily construct a schedule that results in the minimal buffer requirement of 50 but is not a single appearance schedule. Such a schedule is demonstrated in Table 1, where the first row lists the schedule, or the actor firings, and the $e1$ and $e2$ rows show the number of tokens on edge $e1$ and $e2$ respectively after a node has been executed from the schedule. We see that the maximum number of tokens on edge $e1$ is 20 and 30 on edge $e2$, which gives the total buffer memory requirement of 50.

From a Rialto program representing the SDF graph, the compiler produces the following C code (using the more compact code generation approach):

```
enum State {ABC0=1,ABC1=2,ABC2=4,ABC3=8,
            ABC4=16,ABC5=32,ABC6=64,
            ABC7=128,ABC8=256,ABC9=512,
            ABC10=1024 } S;

for ( S=ABC0; S <= ABC10; S<<=1 )
{
  if (S & (ABC0|ABC3|ABC7) )
    code block for A
  else if (S&(ABC1|ABC2|ABC4|ABC6|ABC8|
            ABC9))
    code block for B
  else
    code block for C
}
```

The code has no duplication of the code blocks, and it uses the minimal amount of buffer memory. It uses about the same amount of program memory as the traditional approaches. We can do a bit of control overhead analysis on the two versions: Let n be the number of items in the schedule and k be the number of distinct nodes in the SDF graph. Our code does n increment operations and about $n + \frac{nk}{2}$ comparisons (n comparisons in the outer loop and about $nk/2$ in the if statements on average, depending on how deep we get before a hit), i.e. about $2n + \frac{nk}{2}$ control operations.

The traditional version does n increments, n comparisons and k initialisations, i.e. about $2n + k$ operations total. These are worst-case calculations, and we will see in the example that follow that they are quite pessimistic in both cases. In our case, we can order the if cases so that the ones with the most elements or:ed together, come first. This way we greatly reduce the number of comparisons needed on average to find the code block to execute.

5. Example: Non-uniform Filterbank

This example originates from [11] and is used by Battacharyya, Murthy and Lee. Figure 8 shows an SDF specification of a non-uniform, near-perfect reconstruction filterbank. The repetitions vector [1] of this graph is

$\mathbf{q}(a, \ldots, A) = [27, 27, 9, 9, 18, 6, 6, 9, 12, 6, 9, 4, 4, 6, 8, 4, 4, 4, 12, 6, 6, 9, 18, 9, 27, 27, 27]$

Summing the vector elements together, we get the total number of invocations $n = 313$, the number of nodes in the graph $k = 27$. The best schedule for this graph, that is the single appearance schedule with the lowest buffer memory requirement, obtained by Battacharyya et al. in [2] is the following:
$(3(3(3ab)dc)(2(3e)fgnj)(3kh))(4(3i)mpl(2o)qr(3s))(3(2tu(3w))(3xv(3yzA)))$

[1]The repetitions vector lists the minimum number of times each node has to be fi red during a period of a valid schedule

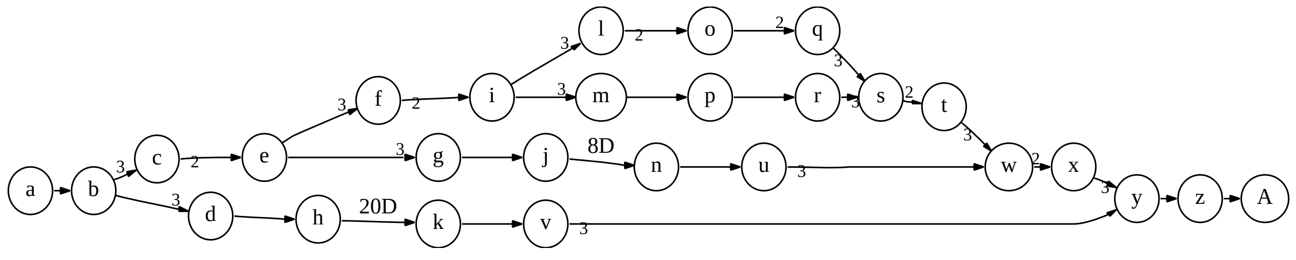


Figure 8. SDF graph for a non-uniform filterbank

They obtain this schedule using a scheduling heuristic they call RPMC to find a single appearance schedule, whose buffer memory requirement is reduced by post-processing using an algorithm called GDPPO. They manage to reduce the buffer memory requirement from 153 to 128. However, the buffer memory lower bound of this graph is 87. Using a very simple minimum-buffer scheduling algorithm, also presented in [2], we find a schedule that achieves the BMLB of 87. The schedule is shown below and as you can see, it is quite big.

$ababababcdeehkvababababcdefgehiijknuababababcdeefghijlm$ $inoopqrsstswwxwuyzyAzyvkAzAababababcdeehababababcdefg$ $eiijlmnoopqrstsswxwwtuyzyAzyvkhxwwAzyAzyAzyvkx$ $wAzyAzyAzyvAzAababababcdeefghiijknuababababcdeehabab$ $abcdefgeijlminoopqrsstswxwwuyzyAzyvkhxAzyAzyAzy$ $vkAzAababababcdeefghiijlmnoopqrstsswwxwtuwyzyAzyvk$ $xwwAzyAzyAzyvxAzyAzyAzyAzA$

According to our analysis, the worst case runtime overhead of code synthesized using the technique of Battacharyya et al. is $2n + k = 2 * 313 + 27 = 653$ operations. A real calculation of the number of loops, and nested loops in the generated code gives 60 initializations and 153 comparisons and increments adding up to 366 operations, which is about half of the worst case.

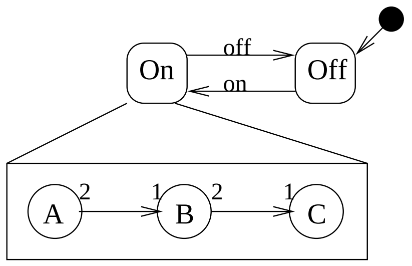
Worst case analysis of our procedure gives $2n + \frac{nk}{2} = 2 * 313 + 313 * 27/2 = 4852$ operations. By combining `or` nodes where all elements in one node are successors of the corresponding ones in the other, as explained earlier, the state space n is reduced from 313 to 251. The compiler finds the following repetition patterns in the schedule ab,cd,fg,lm and $(2o)pqr$. The number of test nodes k is reduced to 20. This would mean 10 comparisons on average per iteration, but ordering the test nodes so that the ones with the most elements come first, we only need to do 5 comparisons on average per iteration, which gives us a runtime overhead of $2 * 251 + 251 * 5 = 1757$ operations per schedule period, which is about one third of the worst case. This is still almost five times more than the traditional technique, which can be quite significant if the nodes only do very simple operations. Let us say that the nodes in the graph (consisting of e.g. low-pass and high-pass filters), do 50 operations per invocation on average. The total number of operations then becomes $50 * 313 + 366 = 16016$ and $50*313+2504 = 18054$ for the two methods; 11% more in our code. We also do get a slightly larger program memory requirement. A compilation of the two pieces of code using GCC with only the control structures resulted in 1720 bytes of object code for the RPMC/GDPPO version and 1764 bytes for our version, 2.1% larger. Since we achieve the BMLB of 87 as opposed to the 128 using RPMC/GDPPO, 30% less, our synthesis approach might be quite useful in certain cases. Notice also that the graph contains two edges with delays, $8D$ and $20D$, which contributes to the buffer memory requirment with 26 tokens (the edges would only need to be of size 1 without delays). If there were no delays, our code would require a buffer memory of 87-26=61, and the other approach at least 102. In that case, our code would have 40% smaller buffer memory. We used about the simplest possible scheduling algorithm, while the RPMC/GDPPO algorithms of Battacharyya et al. are much more complicated.

6. Heterogeneous Models

Embedded systems usually require several description and modeling techniques for different parts of a system. Numerous tools exist for simulation and code generation within the different domains. Also the combination and interaction of the models of computation has been studied by e.g. Lee and Sangiovanni-Vincentelli [10], and the Ptolemy [7] project allows us to simulate heterogeneous models. Usually we still have to resort to different code synthesis tools for the different parts of the system, modeled using different computational models. The generated code is then pieced together using some glue code.

The Rialto language is being developed for the purpose of capturing the semantics of different models of computation and their combinations, and provide a uniform low-level representation for the models that can be optimized for efficient code synthesis.

Let us now look at a hierarchy of computational models, namely an SDF graph with some control modelled using a statechart as depicted in Figure 9. We will use this toy example to demonstrate the use of Rialto in heterogeneous systems, and an alternative SDF policy we could use. The system can be in one of two states, On or Off. While in



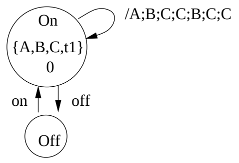
```
upsampler: state
  policy RTC; fifo q; event on, off;
begin
  On: state
    policy RTC;
    t1: trap q.get(off) do l2: goto Off
  begin
    sdf: state
      policy SDF(A,B,C,C,B,C,C);
    begin
      # code block from the first example
    end
  end
  Off: state
    policy rtc;
    t2: trap q.get(on) do l8: goto(On)
  end
end
```

Figure 9. A hierarchy of computational models

the On state, it processes samples as in the first example. While processing, it monitors the off event on its queue and on receiving it, takes the transition to the Off state.

The Rialto code for this system, also shown in the figure, is scheduled on the top-level by the RTC policy, which is a run-to-completion algorithm we have developed for UML statecharts. Also the `On` state is scheduled by the RTC policy, since it is a statechart state but it contains the substate `sdf`, which is scheduled by the SDF policy. The RTC policy of the `On` state mainly takes care of scheduling the `trap` statements according to the statechart semantics. The code would become a bit more compact if we would develop a hybrid scheduling policy that would correctly schedule the traps as well as step through the SDF schedule. This policy would be assigned to the `On` state and the `sdf` substate would not be needed. It is probably wiser to separate the concerns as done in the example though.

The Rialto model is executed by repeatedly calling the RUN function of the topmost statement, which means we call the RTC policy of the `upsampler` state. The program can be translated into the FSM in Figure 10 a). The SDF code blocks `A;B;C;C;B;C;C;` are executed in one atomic step and between executions, the transition to the `Off` state can be taken, but not between node executions in the SDF graph.

a)


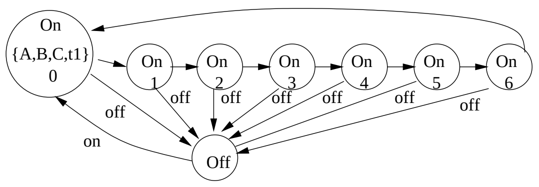
b)


Figure 10. Computation as FSMs, using a) an atomic SDF policy b) and interruptible SDF policy

From this representation we would get the C code below, where the simple FSM is implemented in the `step` function, that calls the `ABCCBCC_P` procedure when it is in the `On` state and no `off` event is observed. The `ABCCBCC_P` procedure contains the code we showed before for first simple SDF example.

```
void ABCCBCC_P(){
  # code as in the first example
}
void step() {
  enum State {Off, On};
  if (S==Off && q.get(on) )
    S = On;
  else if (S==On && q.get(off) )
    S = Off;
  else
    ABCCBCC_P();
}
```

An alternative SDF policy for Rialto, works on a lower level of atomicity, allowing control actors higher up in the hierarchy to interrupt the computation of the SDF schedule. Applying this scheduling policy to the previous example allows the transition to the `Off` state to be triggered between the execution of each node in the schedule. The Rialto program then translates into the FSM in Figure 10 b).

The optimized S-GRAPH representation of this FSM is shown in Figure 11. The S-GRAPH technique has reduced the number of event tests (`q.get(off)`) and transitions (`S:=Off`), as well as removed duplication of the SDF node code blocks. This graph contains 12 nodes, while in its unreduced form it contained 35, a reduction of.

7. Automatic SDF to Rialto Translation

We want to be able to make Rialto totally transparent so that the designer can draw models in an editor, and get

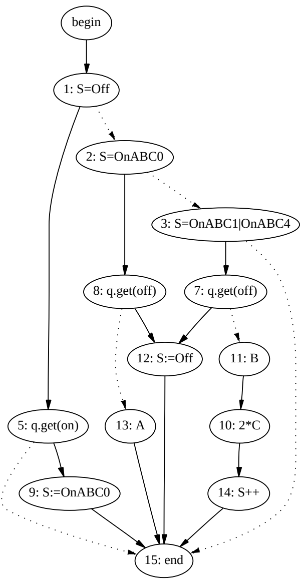


Figure 11. The S-GRAPH for the statechart/SDF model

target code generated for him. For this reason we are working on an SDF profile for the System Modeling Workbench (SMW) [13]. The SMW is a tool for model creation, navigation, manipulation and storage and is based on metamodeling using the MOF standard [12] defined by the Object Management Group (OMG).

Since the SMW is based on metamodeling, profiles for new modelling languages can be rapidly added through a metamodel of the modelling language. In the case of UML, we can obtain the metamodel directly from the object management group in charge of the standardisation of UML; however, a metamodel for SDF had to be created by ourselves. Creating a metamodel for SDF models, is all that is needed for us to be able to textually create and browse SDF models. In order to draw the SDF graphs we need to design a graphical editor for SDF graphs.

We can easily browse the SMW models using python scripts, and for the case of SDF, we could develop scripts to do well formed checks on the graphs, as well as calculate static schedules and buffer sizes for the models. As the last and simplest step, comes the Rialto generation from the SDF graphs.

8. Conclusions and Future Work

We have presented an approach for code synthesis from synchronous dataflow graphs using S-Graphs, implemented in the Rialto compiler. The Rialto approach can also be used for other models of computation, as well as combinations of heterogeneous models. The benefit of our approach for SDF compilation, is that it always results in code with no duplication of actor code blocks regardless of the schedule, also when a single appearance schedule cannot be found. This means that we can focus on minimum buffer memory scheduling. The size of the produced code is about the same as the traditional approaches, but we can achieve smaller buffer memory requirement in cases where a BMLB schedule cannot be found. The drawback of our method is that it results in more runtime control overhead than the usual methods. This becomes significant when the nodes perform only simple calculations, which they very well may do in DSP applications. Our approach could also be used without the Rialto language.

It could perhaps be possible to improve our method by developing special scheduling algorithms for our purposes. The actor invocations could perhaps be organized so that more repetition patterns could be found by the S-Graph optimization steps.

The Rialto compiler is very much on a prototype level, and used only for experiments at this time. We have a simple SDF profile and editor within the SMW framework, from which Rialto code can be generated; this is also work under progress and is still quite limited.

Acknowledgements The author wishes to thank Johan Lilius for useful feedback on the paper and gratefully acknowledges financial support for this work from the Nokia foundation.

References

[1] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. L. Sangiovanni-Vincentelli, and K. Suzuki. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach*. Kluwer Academic Publishers, 1997.

[2] S. Bhattacharyya, P. Murthy, and E. Lee. *Software Synthesis from Dataflow Graphs*, volume 47. Kluwer Academic Publishers, 1996.

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous datafbw specifi - cations. *Journal of VLSI Signal Processing*, 21:151–166, 1999.

[4] D. Björklund and J. Lilius. A language for multiple models of computation. In *Symposium on Hardware/Software Codesign 2002*. ACM, 2002.

[5] D. Björklund, J. Lilius, and I. Porres. Towards effi cient code synthesis from statecharts. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *pUML Workshop at UML2001*, october 2001.

[6] D. Bjrklund and J. Lilius. A unifi ed approach to code generation from behavioral diagrams. In *Forum on specifi cation and design languages (FDL'03)*, 2003.

[7] E. L. et.al. UC Berkeley. The Ptolemy project. http://ptolemy.eecs.berekely.edu/.

[8] R. M. Karp and R. Miller. Properties of a model for parallel computations: Determinacy, termination, queuing. *SIAM Journal of Applied Math*, 14(6), November 1966.

[9] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), September 1987.

[10] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Tran. on CAD*, 17(12), 1997.

[11] K. Nayebi, T. Barnwell, and M. Smith. Nonuniform filter banks: A reconstruction and design theory. *Transactions on Signal Processing*, 41(3), March 1993.

[12] OMG. Meta Object Facility, version 1.4. available at http://www.omg.org/, April 2002.

[13] I. Porres. A toolkit for manipulating UML models. Technical Report 441, Turku Centre for Computer Science, 2002.