-- *Proc. of ICASSP '97, Munich, Germany, April 1997* --

CODE GENERATION BY USING INTEGER-CONTROLLED DATAFLOW GRAPH

Takashi Miyazaki and Edward A. Lee *
Information Technology Research Laboratories, NEC Corporation
1-1, Miyazaki, 4-chome, Miyamae-ku, Kawasaki, Kanagawa, 216, Japan
miyazaki@dsp.cl.nec.co.jp
* Dept. of EECS, University of California, Berkeley
518 Cory Hall, #1770, Berkeley, CA 94720-1770, USA
eal@eecs.berkeley.edu

ABSTRACT

Integer-Controlled Dataflow (IDF) and its code generation applications in Ptolemy are presented. In IDF graphs, which specify data processing systems, data token flow is controlled by integer control tokens and states of actors at run-time. The firing order of actors (schedule) is determined at compile-time, however, the actors are conditionally activated at run-time. This static schedule contributes to effective simulation of systems. IDF supports code generation. This enables code generation from program graphs that include conditional jumps, loops and repetitions, and greatly improves the practical usability of the program synthesis in Ptolemy.

1. INTRODUCTION

Ptolemy [1] is a framework for simulation, prototyping and software synthesis for heterogeneous systems. In Ptolemy, a system is specified by a dataflow graph in which nodes represent computational actors and data token flow between them along the arcs of the graph. Algorithms with control flow that is completely deterministic can be effectively represented by using the synchronous dataflow (SDF) model of computation [2]. In SDF graphs, each actor consumes and produces a constant number of tokens at every firing. The advantage of the SDF model is that it is possible to determine the execution order of actors (schedule) and memory requirements at compile-time. However, data-dependent decision-making at run-time is required in many digital signal processing algorithms. Dynamic dataflow (DDF) [4, 5] is a data-driven model that includes asynchronous operations. The DDF model is usable, but the overhead of run-time scheduling is excessive.

To preserve the compile-time scheduling properties of SDF but permit data-dependent execution, Boolean-controlled dataflow (BDF) [6, 7] was developed. The BDF model of computation extends the SDF model to permit data movement to depend on the values of certain Boolean tokens in the system. The BDF model is successfully applied to simulation and C program synthesis in Ptolemy. Limiting control variables to binary values, however, is overly restrictive. A generalization to integer control variables has been proposed [8].

In this paper, integer-controlled dataflow (IDF) and its code generation implementation in Ptolemy are presented. The IDF graphs, which include IF, CASE, REPEAT and LOOP control structures, support not only simulation but also code generation. C and DSP assembler programs with the IDF structure can be synthesized.

2. INTEGER-CONTROLLED DATAFLOW GRAPH

2.1. IDF Control

A digital signal processing system is described as a dataflow graph in Ptolemy. Data token flow in the IDF graph is controlled by IDF actors. The IDF actor (Fig. 1) evaluates integer values of tokens received from its control port and its own internal state to decide its behavior, such as input and output port selection and data processing at run-time.

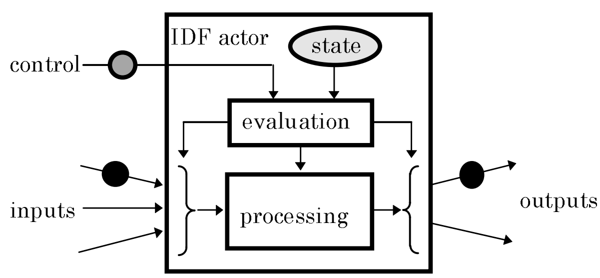

Fig. 1 IDF actor

IDF is derived from BDF in the Ptolemy class hierarchy in order to fully utilize BDF scheduling techniques to control execution of actors in IDF. In BDF, control of token flow at a conditional port of an actor is a function of the value of a Boolean control token. An input conditional port will either consume a token, or consume no token, depending on the control. An output conditional port will either produce a token or not produce a token.

The behavior of IDF actors newly implemented in Ptolemy is managed by an integer control token and a state variable of the actor. To use BDF conditional ports, a decision function is introduced into the IDF dataflow control mechanism as shown in Fig. 2. The decision function evaluates a function of the integer values of a token from a control port and a state of the actor, and returns the Boolean-valued result. A conditional port (either an input or an output) is activated depending on the result of the decision function. The decision function is a user-defined function programmed in the host language. The combination of decision functions and conditional ports makes it possible to create a variety of IDF actors that control token flow.

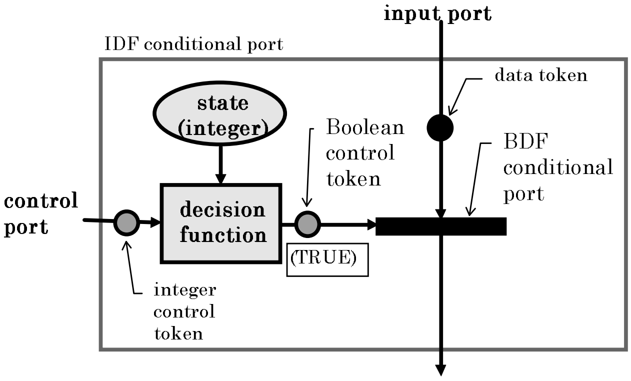

Fig. 2 IDF dataflow control

2.2 IDF Actor

CASE-BEGIN, CASE-END, REPEAT-BEGIN, REPEAT-END and LOOP are a basic set of IDF actors to build IDF graphs. CASE-BEGIN and CASE-END actors work as a switch and a selector, respectively. The CASE-BEGIN actor passes input data tokens to one of output ports selected by a control token. The CASE-END actor receives input data tokens from the selected input port, and send them to the output port. A pair of CASE-BEGIN and CASE-END actors in Fig. 3 form a case structure equivalent to switch-case statements in C programs. In this example, data tokens go through one of three data paths chosen by the case control token. It is obvious how CASE-BEGIN and CASE-END can be implemented with the structure in Fig. 2

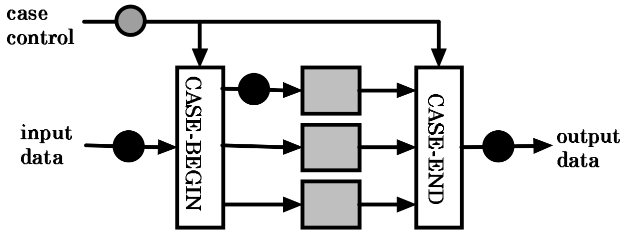

Fig. 3 Case actors

REPEAT-BEGIN and REPEAT-END actors are used for multiphase execution. The REPEAT-BEGIN actor receives a control token (the desired number of repetitions) and an input data token in its first phase, and sends an output token once on each subsequent phase of a cycle. The length of a cycle is determined by the integer control token. The REPEAT-END actor receives input data tokens on every phase of its cycle, but sends an output token only in the last phase. The REPEAT-BEGIN and REPEAT-END actors hold a repeat count as a state, therefore, the control token is required only in the first phase. These REPEAT actors are subtly different from multi-rate actors in SDF, because each repetition is counted as one iteration in the actor execution schedule.

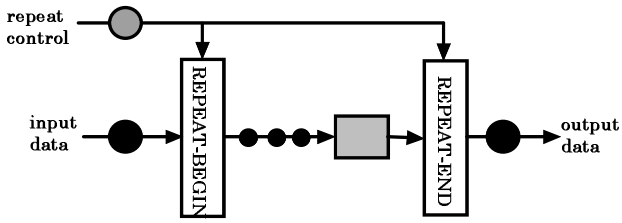

Fig. 4 REPEAT actors

The LOOP actors in Fig. 5 realizes iteration in dataflow graphs. The LOOP actor receives the number of the loop count and an input data token at the beginning of the iteration. The data token cycles around the loop path repeatedly up to the loop counts, and passes out from the output port at the end of iteration. The LOOP actor keeps the

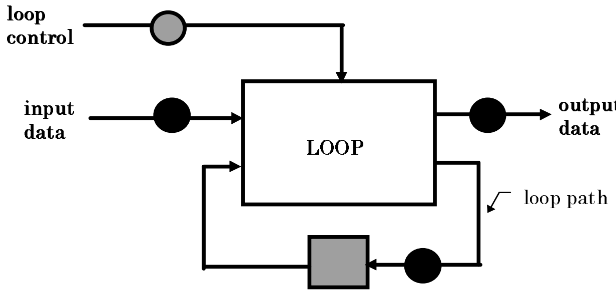

Fig. 5 LOOP actor

iteration count as its state; therefore, it runs by itself after the number of iteration is set.

2.3 IDF Graph

IDF actors such as IF, CASE, REPEAT and LOOP attain flexible system expression. SDF actors are also available in IDF graphs; therefore, a variety of systems can be easily described. A simple (toy) example of an IDF graph is shown in Fig. 6. This graph has a CASE structure. IIDUniform, Abs and QntBtsInt actors generate control tokens. Source data from the Const actor are switched to one of four Gain stars by CASE actors. The switching depends on the value of the control tokens. In the simulation, the graph computes data iteratively according to its execution schedule, and the results are displayed in Xgraph. The combination of static scheduling at compile-time and IDF flow control at run-time greatly contributes to efficient simulation on Ptolemy.

3. CODE GENERATION AND APPLICATION

The IDF capability also supports code generation [9] in Ptolemy. Programmers edit program graphs composed of existing SDF code generation actors and IDF code generation actors such as CASE-BEGIN, CASE-END, REPEAT-BEGIN, REPEAT-END and LOOP. The code generation procedure is as follows. First, the scheduler makes a list that details the firing order of actors (List 1 shows the Ptolemy representation of such a schedule for the program in Fig. 6). Then, buffer size and memory allocation are determined. This is possible, because execution order of the actors is determined at compile-time. Finally, the code is generated by concatenating code blocks in the firing order. The decision functions of IDF actors are defined as subroutines in the target language. When conditional branch statements are found in the schedule during code generation, they are replaced by a call to the decision-making subroutine and a conditional branch instruction in the target language (List 2). To make generated programs compact, IDF actors can share the decision-making subroutines. Code synthesis of C and Motorola 56000 DSP assembler programs from the IDF program graphs is currently attained.

The IDF graphs can effectively describe a variety of digital signal processing block diagrams. For example, in video coding algorithms, such as MPEG and H.26x series, encoding mode is dynamically selected to get better compression result. The mode switching is represented by using CASE actors. REPEAT actors are useful to express iterations like the block-by-block process. Support of the IDF program graphs significantly extends application range of the code generation in Ptolemy. Especially, the capability of DSP program synthesis stimulate reuse of assembler program libraries and helps to design DSP based systems.

The scheduling of the IDF graph relies on the BDF techniques. It is known that the static scheduling of BDF graphs is not always possible. Indeed, it is undecidable for any given graph whether a static, bounded memory schedule can be constructed [7]. Nonetheless, experience with BDF indicates that most practical applications yield static scheduling. In any case, the set of applications that can be statically scheduled is much larger than the set that can be described with SDF.

4. CONCLUSION

IDF and its code generation applications in Ptolemy are presented. The IDF model of computation is built on BDF with the introduction of a decision function. The IDF schedule is determined at compile-time, and actors conditionally run at run-time. IDF graphs, which support CASE, REPEAT and LOOP, have capability to specify a variety of DSP systems. These features contribute to effective and efficient simulation. IDF supports code generation. This enables code generation from program graphs that include conditional jumps, loops and repetitions, and greatly improves the practical usability of the program synthesis in Ptolemy.

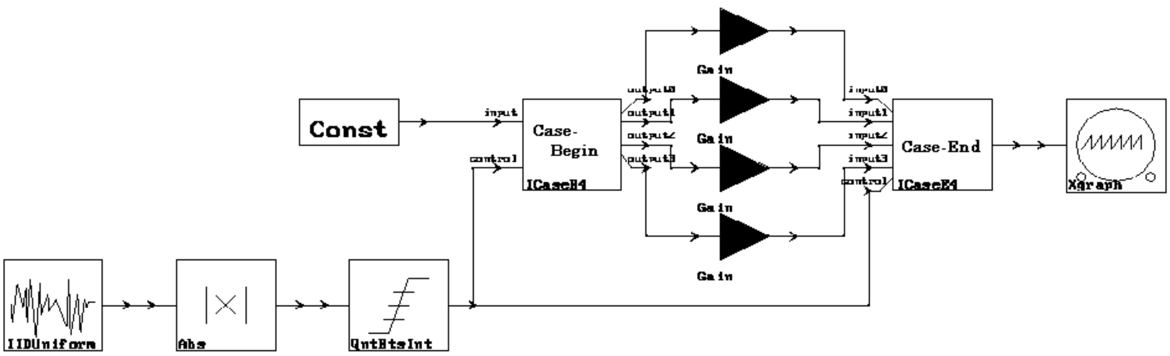

Fig. 6 IDF Graph with case actors

REFERENCES

[1] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," International journal of Computer Simulation, special issue on Simulation Software Development, vol. 4, pp. 155-182, 1994.
[2] E. A. Lee and D. G. Messerschmit, "Synchronous data flow," Proceedings of the IEEE, vol. 75, no. 9, pp. 1235-1245, 1987.
[3] E. A. Lee, "Consistency in Dataflow Graphs," IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No.2, April 1991.
[4] D. G. Messerschmitt, "Structured Interconnection of Signal Processing Programs," Globecom, Atlanta, Georgia, 1984.
[5]D. G. Messerschmitt, "A Tool for Structured Functional Simulation," IEEE Journal on Selected Areas in Communications, vol. SAC-2 no. 1, 1984.
[6] J. Buck and E. A. Lee, "Scheduling Dynamic Dataflow Graphs With Bounded Memory Using the Token Flow Model," Proc. Of ICASSP'93, 1993.
[7] J. Buck, "Scheduling Dynamic Dataflow Graphs With Bounded Memory Using the Token Flow Model," Memorandum No. UCB/ERL M93/69 (Ph.D. Thesis), EECS Dept., University of California, Berkeley, September 1993.
[8] J. T. Buck, "Static Scheduling and Code Generation from Dynamic Dataflow Graphs with Integer-Valued Control Systems," Proc. of IEEE Asilomar Conf. on Signals, Systems, and Computers, Oct. 31, 1994.
[9] J. L. Pino, S. Ha, E. A. Lee and J. T. Buck, "Software Synthesis for DSP Using Ptolemy," Journal of VLSI Signal Processing, 9, 7-21, 1995.
[10] S. Ritz, M. Pankert, V. Zivojnovie and H. Meyr, "High level software synthesis for the design of communication systems," IEEE Journal on Selected Area in Communications, pp. 348 - 358, Apr. 1993.
[11] M. Willems, M. Pankert and S. Ritz, "Fine grain code synthesis within a block diagram oriented code generation environment," Proc. of ICASSP, Detroit, 1995.


```
{ fire case4.demo.IIDUniform1 }
{ fire case4.demo.Abs1 }
{ fire case4.demo.QntBtsInt1 }
{ fire case4.demo.auto-fork-node1 }
{ fire case4.demo.Const1 }
{ fire case4.demo.ICaseB41 }
{ if(case4.demo.ICaseB41.SUB_CntlEq0_0(demo_auto-fork-node1_output#2,statOutput0)) { { fire case4.demo.Gain1 } } }
{ if(case4.demo.ICaseB41.SUB_CntlEq1_1(demo_auto-fork-node1_output#2,statOutput1)) { { fire case4.demo.Gain2 } } }
{ if(case4.demo.ICaseB41.SUB_CntlEq2_2(demo_auto-fork-node1_output#2,statOutput2)) { { fire case4.demo.Gain3 } } }
{ if(case4.demo.ICaseB41.SUB_CntlEq3_3(demo_auto-fork-node1_output#2,statOutput3)) { { fire case4.demo.Gain4 } } }
{ fire case4.demo.ICaseE41 }
{ fire case4.demo.Xgraph1 }
```

List 1 Run schedule of actors for the IDF graph with case actors

```
;------ Beginning of conditional branch (begin-if) [Depth = 1] ------
;   if(case4.demo.ICaseB41.SUB_CntlEq0_0(control,statOutput0) != 0)
        move    x:1,a               ; load 'control' to A register
        move    y:1,b               ; load 'state' to B register
        jsr     SUB_CntlEq0_0       ; call an evaluation function
        jne     IFBRANCH_10         ; branch if result of the eval. func. is FALSE (Z flag == 0)
; code from star case4.demo.Gain1 (class CG56Gain)
        clr     a
        move    a,x:3
;------ End of conditional branch (end-if) [Depth = 1] ------
IFBRANCH_10
```

List 2 DSP assembler program generated for conditional run of an actors