INTEGER-CONTROLLED DATAFLOW IN PTOLEMY

by

Takashi Miyazaki

Memorandum No. UCB/ERL M97/21

19 March 1997

INTEGER-CONTROLLED DATAFLOW IN PTOLEMY

by

Takashi Miyazaki

Memorandum No. UCB/ERL M97/21

19 March 1997

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Integer-Controlled Dataflow in Ptolemy

Takashi Miyazaki †

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley

ABSTRACT

In this report, integer-controlled dataflow (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 static and conditional, so that memory requirement is determined at compile-time. 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.

---

† Visiting Industrial Fellow from Information Technology Research Laboratories, NEC Corporation, from September 1, 1995 to August 31, 1996.

1. INTRODUCTION

Ptolemy [1] is a framework for simulation, prototyping and software synthesis for heterogeneous systems. In Ptolemy, a system is specified by dataflow graphs in which nodes represent computational actors and data tokens 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, overly restrictive. A generalization to integer control variables has been proposed [8].

In this report, 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

2.1. Model of Computation

Concept of Integer-controlled dataflow is originally presented in [8]. In this report, the IDF model of computation implemented in Ptolemy is explained.

In IDF, dataflow is controlled by integer values of a control token and / or a state of an actor (Star). An IDF actor has a control port which receives a control token and its own internal state, as shown in Fig. 1. Behavior of an IDF actor, such as data input port selection, computation in the actor and output port selection, is determined by both or one of the integer values of the control token and the internal state at run-time.

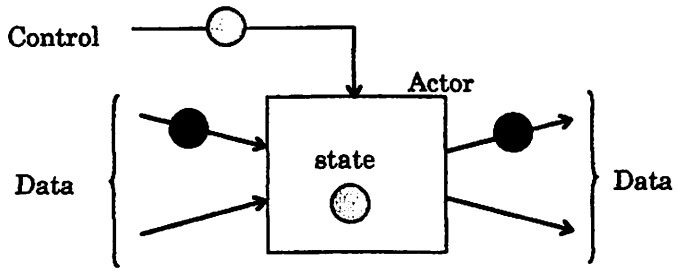

Fig. 1 IDF Actor

IDF dataflow graphs can describe a variety of dataflow control, for instance, BRANCH, CASE, REPEAT and LOOP.

(1)BRANCH (IF) and CASE

BRANCH is a general case of IF. Fig. 2 and Fig. 3 show two cases of BRANCH, such as data switching and data selection, respectively. CASE-Begin and CASE-End are IDF actors. CASE-Begin actor chooses an output port by the integer value of the control token, and sends the input data token to the selected output port. CASE-End forms a reverse structure of CASE-Begin. CASE-End actor receives an input token from the selected input port, and send it to the output port.

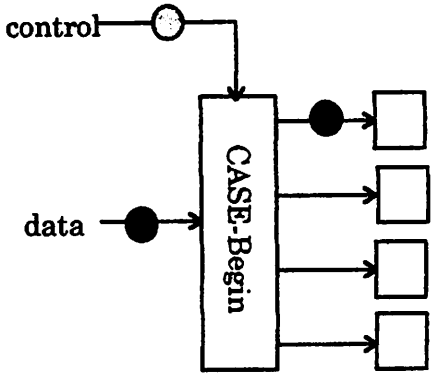

Fig. 2 CASE-Begin

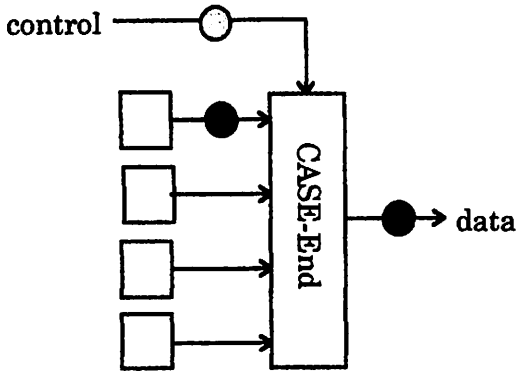

Fig. 3 CASE-End

A CASE structure shown in Fig. 4 is a counterpart of switch-case statements in C programs. Two IDF actors, CASE-Begin and CASE-End, forms the CASE structure, and controlled by the same integer control token. In this example, data tokens go through one of four conditional branches.

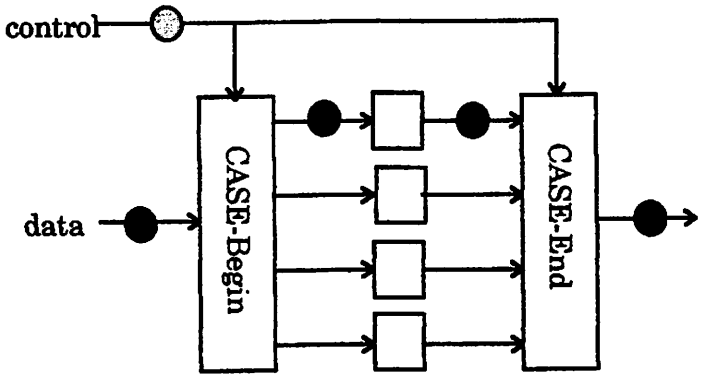

Fig. 4 CASE

(2)REPEAT

REPEAT actors express repetition of actor execution in dataflow graphs. REPEAT-Begin and REPEAT-End actors shown in Fig. 5 and Fig. 6 work as controllers of beginning and ending of repetitions, respectively. REPEAT-Begin actor reads the number of repetition and an input data token at the beginning of repetition, and sends out output data tokens iteratively in the number of repetition count. REPEAT-End actor reads the number of repetition at the beginning of repetition. It receives input data tokens iteratively, and sends out an output token at the end of repetition. REAET actors are different from multi-rate actors, because each repetition is counted as one iteration of actor execution. Actors which follow the REPEAT-Begin actor or are followed by the REPEAT-End actor may be executed at each repetition of REPEAT actors.

Output data tokens of REPEAT-Begin and input data tokens of REPEAT-End are defined by users, therefore, REPEAT actors are customized. Current repetition count is stored as an internal state. For example, an array data and its elements may be input and output tokens of REPEAT-Begin. In the video processing, an image data is an input, and block data are output.

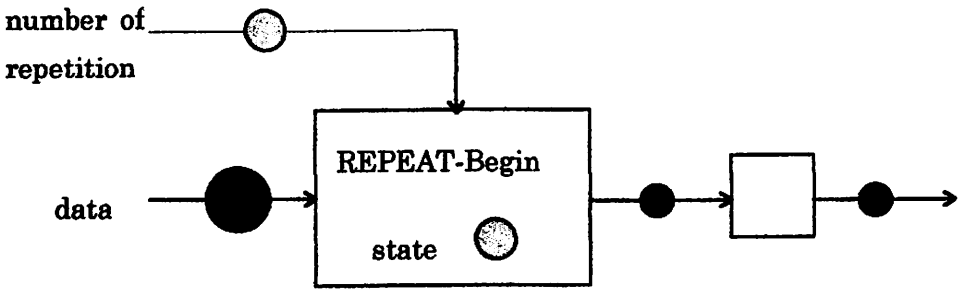

Fig. 5 REPEAT-Begin

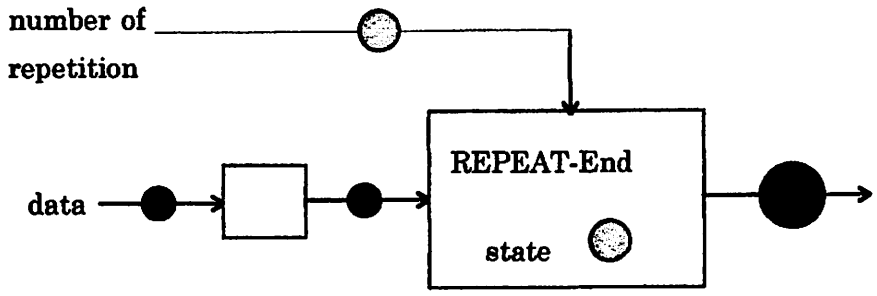

Fig. 6 REPEAT-End

REPEAT actor, which combine REPEAT-Begin and REPEAT-End, expresses a repeat structure in dataflow graphs, as shown in Fig. 7. This actor receives the number of repetition and an input data token at the beginning of repetition. During repetition, data tokens go through the loop path, and an output data token are put out from the output port.

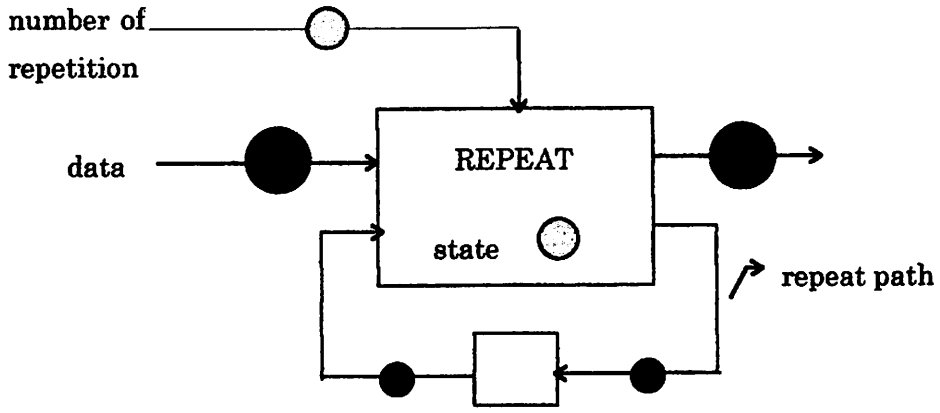

Fig. 7 REPEAT

(3)LOOP

LOOP actor realizes a loop structure in dataflow graphs. Fig. 8 shows an example of the loop. LOOP actor receives the number of loop counts and an input data token at the beginning of loop. The data token goes on the loop path iteratively up to loop counts, and gets out from the output port at the end of loop.

The differences between LOOP and REPEAT are that one data token is repeatedly processed by actors on the loop path in the loop structure, on the other hand, tokens divided from an original input data token are put on the repeat path, and summed up to be an output data token in the REPEAT.

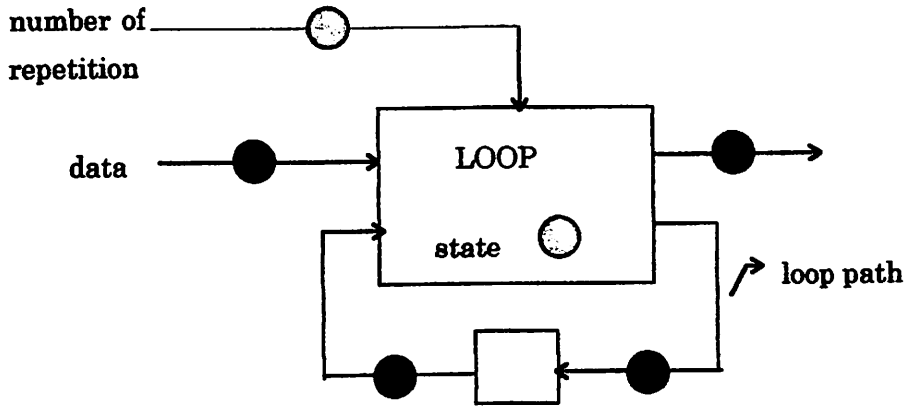

Fig. 8 LOOP

2.2. Scheduling and Control of Dataflow in Ptolemy

IDF is implemented as one of model of computation (Domain) in Ptolemy. The IDF Domain is derived from the BDF Domain. Therefore IDF inherits most of BDF properties.

IDF uses the BDF scheduler as its own scheduler to determine static execution order of actors. Therefore, the performance of actor scheduling in IDF is the same of BDF. Details of BDF is described in [7].

In order to realize IDF dataflow control on the BDF dataflow control mechanism, a decision function is attached to each conditional input/output port. The decision function evaluates integer values of an integer control token from a control port and an internal state, and return TRUE or FALSE value. The TRUE/FALSE value is used to control conditional input/output ports.

The decision function is user-defined. As described, the decision function estimates two values of integer control token from a control port and an integer internal status in an actor. The control token is used to manage the IDF actor from outside. The internal status is used to manage the IDF actor from inside. For example, CASE-Begin and CASE-End actors are controlled by only the control token. On the other hand, REPEAT and LOOP actors are controlled by only the internal status, which stores loop counts.

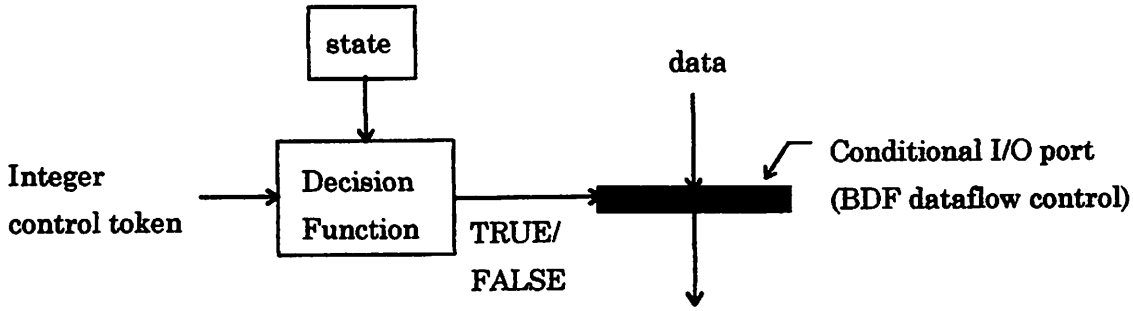

Fig. 9 Mechanism of Dataflow Control in IDF

3. Simulation

3.1. Default-IDF Target

The default-IDF target supports the IDF model of computation. It must be used when IDF stars are present in the dataflow graph. It can also be used with graphs that contain only SDF stars. It does not support graphs with BDF stars, however, the graphs can be re-used, when the BDF stars are replaced with the same functional IDF stars. ISelect and ISwitch stars are used for this purpose. The default-IDF target supports single processors. The default-idf target has the same parameters as the default-bdf target.

3.2. IDF Stars

IDF stars are used for conditionally routing data and looping and repeating. IDF stars require the IDF target, because the stars require the IDF scheduler and IDF dataflow control mechanism.

(1)ISelect

If the value on the control line is nonzero (TRUE), trueInput is copied to the output; otherwise, falseInput. This star is equivalent to the Select star in BDF stars.

(2)ISwitch

This star switches an input token to one of two outputs, depending on the value of the control input. If the value on the control line is nonzero (TRUE), input is copied to the trueOutput; otherwise, falseOutput. This star is equivalent to the Switch star in BDF stars.

(3)ICaseB4

This star is an example of IDF stars. This star switches an input token to one of 4 outputs, depending on the value of the control input. Modulo of 4 of the control input is currently implemented on evaluation functions to switch the input token.

(4)ICaseE4

This star is an example of ITDF stars. This star selects one of 4 inputs, and copies it to the output, depending on the value of the control input. Modulo of 4 of the control is currently implemented on evaluation functions to select the input token.

ICaseBn and ICaseEn stars can form a case structure of dataflow corresponding to switch-case structure in C/C++ programs. See a demo of ICase4-demo

(5)ILoop

This star forms a loop structure. At the beginning of loop process, the stare receives a loop count from the set port, and a data token from input. During the loop process, the data token is put out from loopFor port, and put in from loopBack port. Stars on the path between loopFor and loopBack represent the loop process imposed on the looped token. At the end of the loop, the processed token is sent out from output port.

CntlInI, cntlInO and cntlOut ports are used only to determine star schedule by the IDF (BDF) scheduler. Its connection must be the same as a loop structure demonstration of ILoop-demo. This restriction comes from the BDF scheduler, which is an original of the IDF scheduler. At run time, tokens from the control port are not necessary, and are discarded.

(6)IRepeatB

This star is an example of IDF stars. This star receives one token, and sends the token out repeatedly. The repeat count depends on set port value. If the set port value is N, the input token is sent to output N times. The input and set ports receives a token respectively at the beginning of repeat. The control port is used only to determine a star schedule by the IDF scheduler. At run time, tokens from
the control port are not necessary, and are discarded.

(7)IRepeatE

This star is an example of IDF stars. This star receives tokens repeatedly from the input port, sums them, and sends the result out from the output port. The repeat count depends on the value of the set port. If the set port value is N, N input tokens are received, and one result is sent to the output port. The set ports receives a token at the beginning of repeat. The output port sends a token out at the end
of the repeat. The control port is used only to determine star schedule by the IDF scheduler. In the execution, tokens from the control port are discarded.

The IRepeatB and IRepeatE stars can form a repeat process, in which each token produced by IRepeatB is processed by functional stars on the datapath between IRepeatB and IRepeatE stars. The IRepeatE star collects the processed tokens.

3.3. Demos

(1)CASE

The CASE demo shown in Fig. 10 is an example of a case structure in dataflow graphs. Source data from the Const star are switched to one of four Gain stars. The switching depends on the value of the control tokens from the IIDUniform star. The ICaseB4 and ICaseE4 stars choose one of four data path. Two windows appear to show input and output data.

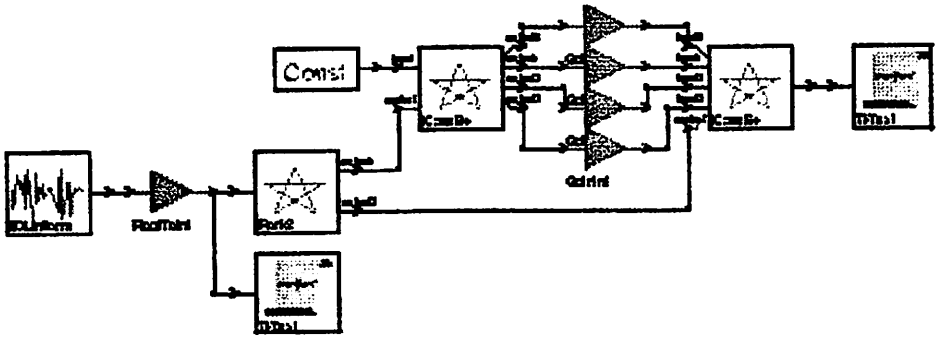

Fig. 10 CASE demo in IDF simulation

(2)REPEAT

The repeat demo is shown in Fig. 11. Two REPEAT actors, IRepeatB and IRepeatE form a repeat loop. These REPEAT stars read the common repetition count at the beginning of repeat. The data path composed of IFork2 and Delay is necessary for the IDF (BDF) scheduler to determine a star schedule, however, at run-time, data on this path is not used. This path should be invisible to avoid unexpected confusion.

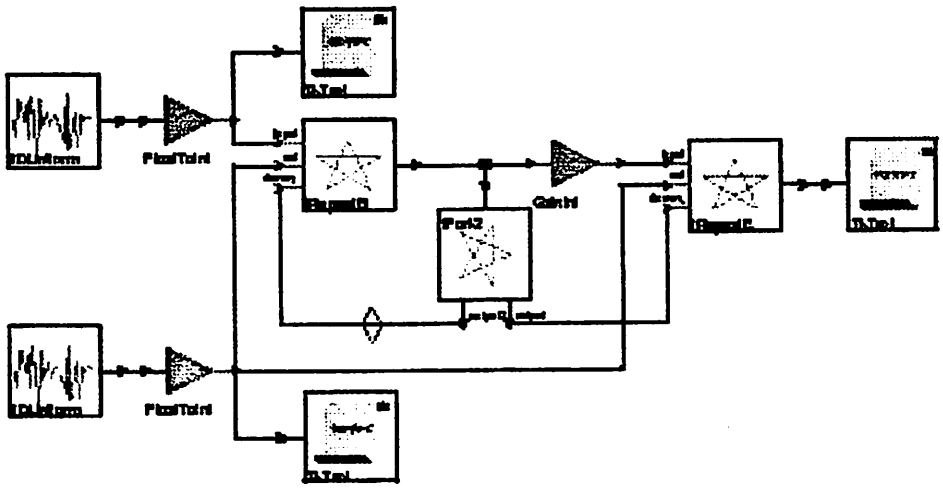

Fig. 11 REPEAT demo in IDF simulation

(3)LOOP

A Loop demo is shown in Fig. 12. In this demo, data is amplified by GainInt Star. The data path composed of IFork2 and Delay is necessary for the IDF (BDF) scheduler to determine a star schedule, however, at run-time, data on this path is not used. This path should be invisible to avoid unexpected confusion.

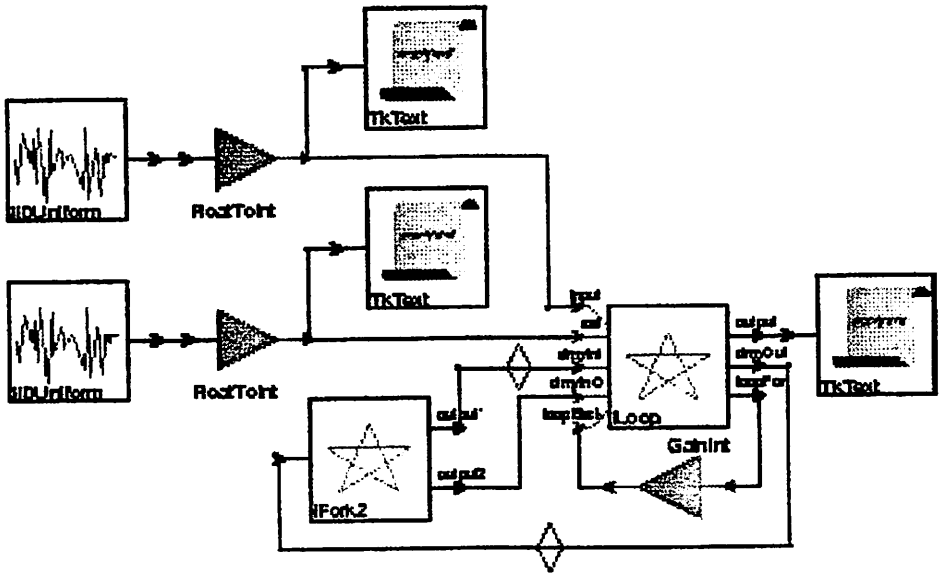

Fig. 12 LOOP demo in IDF simulation

3.4. Writing IDF stars

Description of IDF stars basically obeys to rules of writing other simulation stars. Additional descriptions are required to let the IDF scheduler know dataflow control staffs and to control dataflow at run-time.

For convenience, the ICaseB4 star is shown as an example of writing IDF stars.

```
name    { ICaseB4 }
domain { IDF }
```

The definitions of the star name and domain are common among all stars. The domain name is "IDF".

```
input
{
  name { control }
  type { int }
  desc { control to select one out of conditional outputs }
}
```

Definition of input and output ports is also common among all stars. The star writer must be careful that the control port, whose value is used to determine behavior of a conditional port, must be a type of "int".

```
hinclude { "IdfCntl.h" }
```

Including a header file, "IdfCntl.h", is currently mandatory, except for BDF-like stars

such as the ISwitch and ISelect stars. In the file, a class data used for IDF dataflow control is defined.

```
defstate
{
  name      { statOutput0 }
  type      { int }
  default   { 0 }
  desc      { state variable of output0 }
}
```

States of conditional I/O ports, which are also used to determine behavior of a conditional port, should be defined in the defstate method. Its type is also a type of "int".

```
protected
{
      IdfCntl cStatOutput0;
  ......
}
```

The IdfCntl class is defined in "IdfCntl.h". Each conditional port needs IdfCntl class data to let the IDF scheduler know dataflow control staffs of the conditional port through the IDFPortHole::Set() method. The details will be described later.

```
code
{
  // if modulo 4 of cntl is 0, return 1
  int ISCaseB4_mod4_0(int cntl, int st) { return ((cntl % 4) == 0); }

  }
```

Decision functions to determine behavior of conditional ports are defined here. The function must have two int type arguments. One is for a value of a control port, and the other is for a value of the I/O port status. This function must return TRUE (1) or FALSE (0) value.

```
cStatOutput0.Set((const char*)fullName(),
                 "control",
                 "statOutput0",
                 "ISCaseB4_mod4_0",
                 (EVALFUNC)ISCaseB4_mod4_0,
                 &statOutput0);
```

IdfCntl::Set() method stores dataflow control staffs in its object.

```
void CntlState::Set(const char* starName,
                    const char* cntlName,
                    const char* stateName,
                    const char* funcName,
                    EVALFUNC func,
                    IntState*    stat)
```

starName ··· star name. fullName method may be used.
CntlName ··· control port name
stateName ··· internal state name
funcName ··· decision function name
func ··· pointer to the decision function. The data type is EVALFUNC.
stat ··· pointer to the internal state data

```
output0.setIDFParams(n,control,IDF_TRUE,cStatOutput0,n-1);
```

PortHole::setIDFParams() method sets the relation of this port with associated porthole and dataflow control staffs.

```
void PortHole:: setIDFParams(int n,
                             PortHole control,
                             IDFRelation rel,
                             IdfCntl cStat,
                             int n-1);
```

n ··· number of input tokens
control ··· control port
rel ··· Specify the relation of this port with the result of the evaluation function.
DF_TRUE : produce/consume data only when the result of evaluation function is TRUE.
DF_FALSE : produce/consume data only when the result of evaluation function is FALSE.
CStat ··· IdfCntl data
n-1 ··· buffer size

```
go
{
    int i;
    int n = int(N);
    // read control value, and route input to output depending on it.
    int cntl = int(control%0);
    // do conditional outputs
    if(cStatOutput0.eval(cntl) == IDF_TRUE)
    { for(i = 0; i < n; i++) { output0%i = input%i; } }

}
```

In the go method(), behavior of this actor is defined. In CaseB star, conditions of each conditional port are evaluated, and if the condition is TRUE, the port is activated.

4. CODE GENERATION

4.1. C Code Generation

The IDF model of computation is implemented on code generation for the C programming language. The idf-CGC target, added to the members in the CGC target list, supports program graphs that contain SDF and IDF stars.

4.1.1. idf-CGC Target

The idf-CGC target supports the IDF model of computation. It must be used when IDF stars are present in the program graph. It can also be used with program graphs that contain only SDF stars. It does not support program graphs with BDF stars, however, the program graphs can be re-used, when the BDF stars are replaced with the same functional IDF stars. CGCISelect and CGCISwitch stars are prepared for this purpose. The idf-CGC target supports single processors. The idf-CGC target has the same parameters as the bdf-CGC target.

The idf-CGC program graphs are similar to the bdf-CGC. This is because the IDF dataflow control method originally comes from the BDF. However, IDF program graphs are more concise and explicit than BDF.

4.1.2. CGC/IDF Stars

IDF stars are used for conditionally routing data and looping and repeating. IDF stars require the idf-CGC target, because the stars require the IDF (BDF) scheduler and IDF dataflow control mechanism. Unlike their simulation counterparts, these stars can only transfer single tokens in one firing.

(1)ISelect

If the value on the control line is nonzero (TRUE), trueInput is copied to the output; otherwise, falseInput. This star is equivalent to the Select star in CGC/BDF stars.

(2)ISwitch

This star switches an input token to one of two outputs, depending on the value of the control input. If the value on the control line is nonzero (TRUE), input is copied to the trueOutput; otherwise, falseOutput. This star is equivalent to the Switch star in CGC/BDF stars.

(3)ICaseB4

This star is an example of CGC/IDF stars. This star switches an input token to one of 4 outputs, depending on the value of the control input. Modulo of 4 of the control input is currently implemented on evaluation functions to switch the input token.

(4)ICaseE4

This star is an example of CGC/IDF stars. This star selects one of 4 inputs, and copies

it to the output, depending on the value of the control input. Modulo of 4 of the control is currently implemented on evaluation functions to select the input token.

ICaseBn and ICaseEn stars can form a case structure of dataflow corresponding to switch-case structure in C/C++ programs. See a demo of ICase4-demo

(5)ILoop

This star forms a loop structure. At the beginning of loop process, the stare receives a loop count from the set port, and a data token from input. During the loop process, the data token is put out from loopFor port, and put in from loopBack port. Stars on the path between loopFor and loopBack represent the loop process imposed on the looped token. At the end of the loop, the processed token is sent out from output port.

CntlInI, cntlInO and cntlOut ports are used only to determine star schedule by the IDF (BDF) scheduler. Its connection must be the same as a loop structure demonstration of ILoop-demo. This restriction comes from the BDF scheduler, which is an original of the IDF scheduler. At run time, tokens from the control port are not necessary, and are discarded.

(6)IRepeatB

This star is an example of CGC/IDF stars. This star receives one token, and sends the token out repeatedly. The repeat count depends on set port value. If the set port value is N, the input token is sent to output N times. The input and set ports receives a token respectively at the beginning of repeat. The control port is used only to determine a star schedule by the IDF (BDF) scheduler. At run time, tokens from the control port are not necessary, and are discarded.

(7)IRepeatE

This star is an example of CGC/IDF stars. This star receives tokens repeatedly from the input port, sums them, and sends the result out from the output port. The repeat count depends on the value of the set port. If the set port value is N, N input tokens are received, and one result is sent to the output port. The set ports receives a token at the beginning of repeat. The output port sends a token out at the end of the repeat. The control port is used only to determine star schedule by the IDF (BDF) scheduler. In the execution, tokens from the control port are discarded.

The IRepeatB and IRepeatE stars can form a repeat process, in which each token produced by IRepeatB is processed by functional stars on the datapath between IRepeatB and IRepeatE stars. The IRepeatE star collects the processed tokens.

4.1.3. CGC/IDF demos

(1)if ThenElse

The ifThenElse demo is equivalent to the ifThenElse demo. Difference between the both is that the Switch and Select stars are replaced with the ISwitch and ISelect stars. Results of the both are the same.

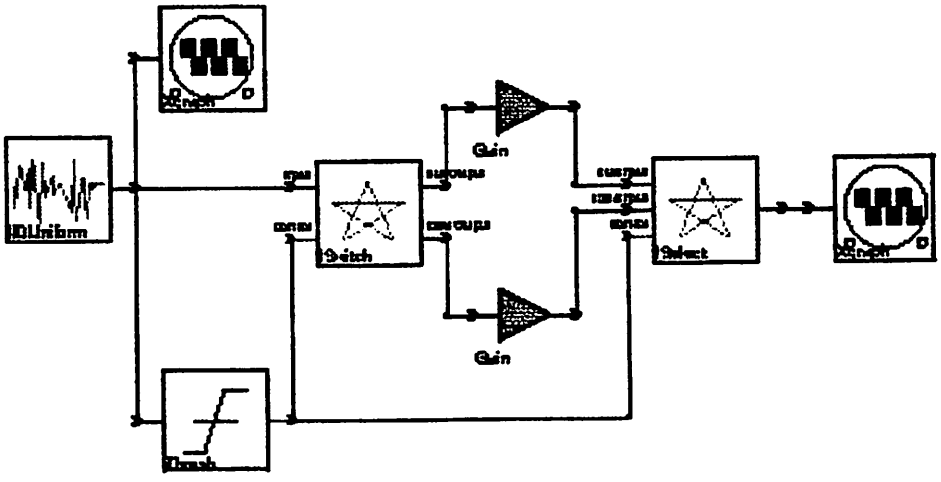

Fig. 13 ifThenElse demo in CGC/IDF

(2)Case

The Case-demo shows an example of a case structure in program graphs. Source data from the IIDUniform star are switched to one of four Gain stars. The switching depends on the value of the control line of the ICaseB4 and ICaseE4 stars. The ICaseE4 star selects data through a gain. Two graphs show input and output data.

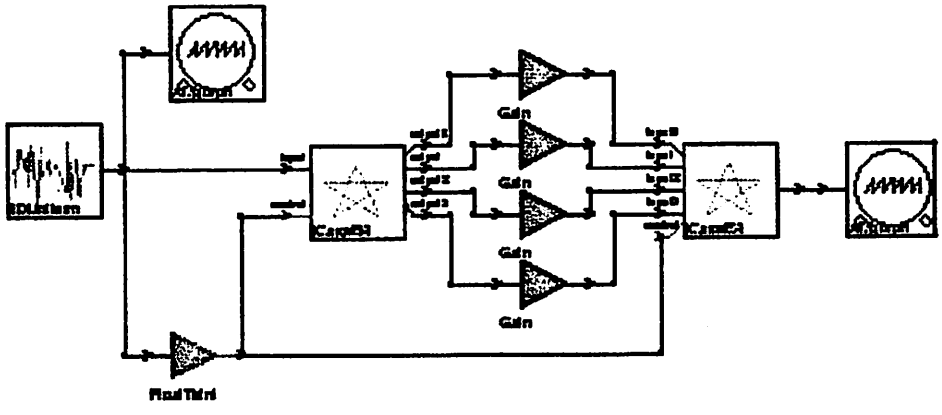

Fig. 14 Case demo in CGC/IDF

(3)Loop

The Loop demo shows an example of a loop structure in program graphs. The ILoop star read a source data from the IIDUniform star at the beginning of a loop process. At the same time, a loop count is also read from the set port. The data go round on the loop path between the loopFor and loopBack ports. At the end of loop, the looped data is put out from the output port. Two graphs show input and output data to/from the loop. CntlInI, cntlInO and cntlOut ports are used only to determine star schedule by the IDF (BDF) scheduler. Its connection must be the same as a loop structure demonstration of

the Loop demo. This restriction comes from the BDF scheduler, which is an original of the IDF scheduler. At run time, tokens from the control port are discarded.

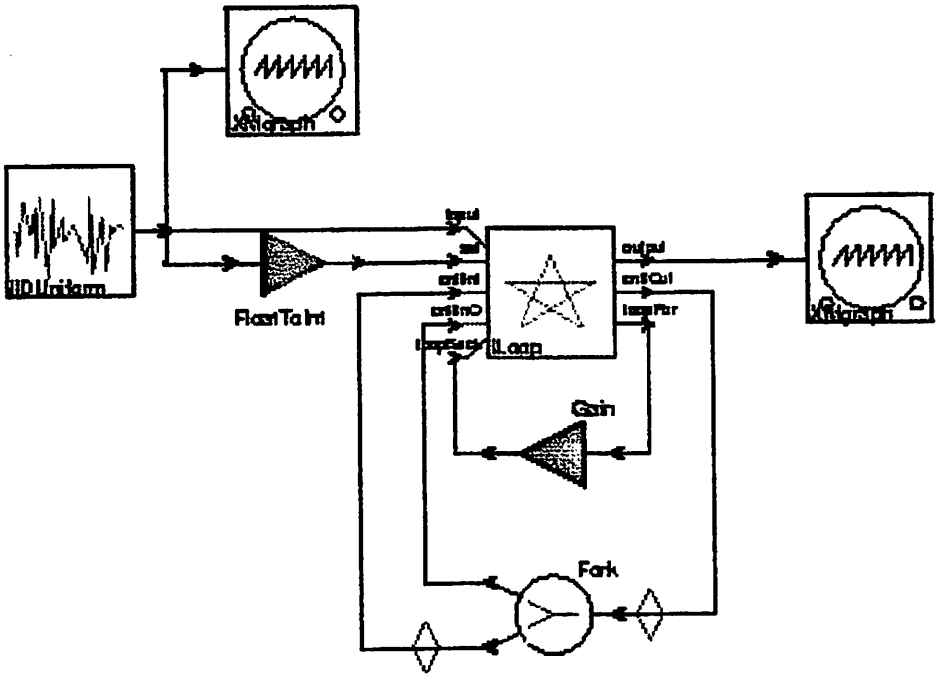

Fig. 15 Loop demo in CGC/IDF

(4)Repeat

The Repeat demo shows an example of a repeat structure in program graphs. The IRepeatB star receives one data token from its input port, and a repeat count from its set port. The star sends the data tokens repeatedly. IRepeatE star sums the tokens through the Gain star. Two graphs show input and output data to/from the repeat. The paths to control ports of the IRepeatB and IRepeatE stars are necessary, because of restriction of the BDF (IDF) scheduler. At run time, tokens on the control path are discarded.

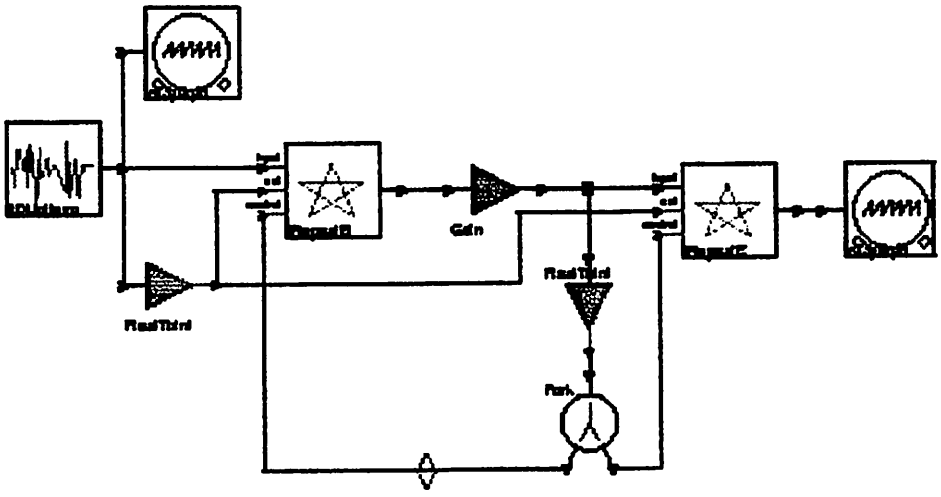

Fig. 16 Repeat demo in CGC/IDF

4.1.4. Writing CGC/IDF stars

Description of CGC/IDF stars also obeys to rules of writing CGC stars. Additional descriptions are required to let the IDF (BDF) scheduler know dataflow control staffs and to control dataflow at run time.

For convenience, the ICaseB4 star is shown as an example of writing CGC/IDF stars.

```
name    { ICaseB4 }
domain { CGC }
```

The definitions of the star name and domain are common among all stars. The domain name is "CGC".

```
input
{
  name { control }
  type { int }
  desc { control to select one out of conditional outputs }
}
```

Definition of input and output ports is also common among all stars. The star writer must be careful that the control port, whose value is used to determine behavior of a conditional port, must be a type of "int".

```
defstate
{
  name        { statOutput0 }
  type        { int }
  default     { 0 }
```

```
    attributes  { A_NONCONSTANT }
    desc        { state variable of output0 }
  }
```

State of conditional I/O ports, which are also used to determine behavior of a conditional port, should be defined in the defstate method. Its type is also a type of "int". Its values would be changed, so its attribute is usually "A_NONCONSTANT".

```
  hinclude { " IdfCntl.h" }
```

Including a header file, " IdfCntl.h", is currently mandatory, except for BDF-like stars such as the ISwitch and ISelect stars.

```
  protected
  {
    IdfCntl cStatOutput0;
    ......
  }
```

The IdfCntl class is defined in "IdfCntl.h". Each conditional port needs CntlState class data to let the IDF scheduler know dataflow control staffs of the conditional port through the IDFPortHole::Set() method. The details will be described later.

```
  codeblock(evalFunction)
  {
    int $sharedSymbol(ICaseB4,ICaseB4_mod4_0)(int cntl, int st)
    { return ((cntl % 4) == 0); }
    .......
  }
```

Decision functions to determine behavior of conditional ports are defined here. The function must have two int type arguments. One is for a value of a control port, and the other is for a value of the I/O port status. This function must return TRUE (1) or FALSE (0) value. Be careful that the function name is defined by using $sharedSymbol() macro. This is because the evaluation functions are written as a global code in generated programs.

```
  method
  {
    name { notone }
    type { int }
    arglist { "(CGCPortHole& port)" }
    code
    {
      return (port.numInitDelays() > 1 || port.far()->numXfer() > 1);
    }
```

```
}
```

This method is used to check the number of particle consumed or produced at portholes, and return FALSE (0) if not one. This is because porthole can consume or produce only one particle when it fires. This is the same as CGC/BDF stars. This method will appear in a setup method of the star, and be necessary to avoid unexpected result.

```
setup
{
  if(notone(input)    || notone(control) ||
     notone(output0) || notone(output1) ||
     notone(output2) || notone(output3))
  {
     Error::abortRun(*this,
        "Non-unity buffers connected to a switch not yet supported");
  }
 (CONTINUED)
```

In the setup method of the star, function of notone() appears, first. This function is described above. It is wise to stop run, if the result of this function is FALSE (0), or unexpected result would worry you.

```
  else
  {
   // make all the buffers overlap.
    input.embed(output0,0);
    .....
```

embed() method is described in "Buffer Embedding" in CGC Domain section in "Ptolemy Programmer's Manual".

```
cStatOutput0.Set((const char*)fullName(),
                 "control",
                 "statOutput0",
                 (const char*)evalFuncSymbol("ICaseB4","ICaseB4_mod4_0"),
                 (EVALFUNC)DUMMY_EVALFUNC,
                 &statOutput0);
```

IdfCntl::Set() method stores dataflow control staffs in its object.

```
void CntlState::Set(const char* starName,
                    const char* cntlName,
                    const char* stateName,
                    const char* funcName,
                    EVALFUNC func,
                    IntState*    stat)
```

starName ··· star name. fullName method may be used.
CntlName ··· control port name
stateName ··· internal state name
funcName ··· decision function name. It is OK to write "DUMMY_EVALFUNC".
func ··· pointer to the decision function. The data type is EVALFUNC. Use the function, evalFuncSymbol(char*, char*), to pass the name. Arguments of the function must be the same as the $sharedSymbol() macro in the evaluation function definition.
stat ··· pointer to the internal state data

```
// set relation
output0.setRelation(DF_TRUE, &control, &cStatInput0);
```

PortHole::setRelation() method sets the relation of this port with associated porthole and dataflow control staffs.

void PortHole::setRelation(DFRelation relation, DynDFPortHole* assoc, CntlState* stat)

relation ... Specify the relation of this port with the result of the evaluation function.
DF_TRUE : produce/consume data only when the result of evaluation function is TRUE.
DF_FALSE : produce/consume data only when the result of evaluation function is FALSE.

```
initCode
{
 // add code of evaluation functions to the code stream
  addGlobal(evalFunction);
}
```

In initCode method, the code block of the evaluation function must be added in the global code section.

The ICaseB4 star does not have go() method. The star writer can freely write codes in go and wrapup methods. The writers must obey to rules of writing IDF stars. Description of go method usually consists of data receive section, state change section and data send section, and these sections appear in this order.

4.2. DSP Code Generation

The IDF model of computation is implemented on code generation for the Motorola 56000 assembler language. The idf-CG56 target, added to the members in the CG56 target list, supports program graphs that contain CG56/SDF and CG56/IDF stars.

The CG56 domain did not have method of conditional execution of stars. This sometimes restricts programmability. Now you can draw flexible program flows of assembler programs.

4.2.1. idf-CG56 Target

The idf-CG56 target supports the IDF model of computation. It must be used when CG56/IDF stars are present in the program graph. It can also be used with program graphs that contain only CG56/SDF stars. The IDF scheduler support the BDF model of computation, so that BDF-like stars also can be used. CG56ISelect and CG56ISwitch stars have similar functions to BDF stars. The idf-CG56 target supports single processors.

4.2.2. CG56/IDF Stars

CG56/IDF stars are used for conditionally routing data and looping and repeating. CG56/IDF stars require the idf-CG56 target, because the stars require the IDF (BDF) scheduler and IDF dataflow control mechanism. Unlike their simulation counterparts, these stars can only transfer single tokens in one firing.

(1)ISelect

If the value on the control line is nonzero (TRUE), trueInput is copied to the output; otherwise, falseInput.

(2)ISwitch

This star switches an input token to one of two outputs, depending on the value of the control input. If the value on the control line is nonzero (TRUE), input is copied to the trueOutput; otherwise, falseOutput.

(3)ICaseB4

This star is an example of CG56/IDF stars. This star switches an input token to one of 4 outputs, depending on the value of the control input. In this example, if control is one, the star send input to the output-1 port

(4)ICaseE4

This star is an example of CG56/IDF stars. This star selects one of 4 inputs, and copys it to the output, depending on the value of the control input.

ICaseBn and ICaseEn stars can form a case structure of dataflow corresponding to switch-case structure in C/C++ programs. See a demo of case4

(5)ILoop

This star forms a loop structure. At the beginning of loop process, the stare receives a loop count from the set port, and a data token from input. During the loop process, the data token is put out from loopFor port, and put in from loopBack port. Stars on the path between loopFor and loopBack represent the loop process imposed on the looped token. At the end of the loop, the processed token is sent out from output port.

CntlInI, cntlInO and cntlOut ports are used only to determine star schedule by the

IDF (BDF) scheduler. Its connection must be the same as a loop structure demonstration of ILoop-demo. This restriction comes from the BDF scheduler, which is an original of the IDF scheduler. At run time, tokens from the control port are not necessary, and are discarded.

(6)IRepeatB

This star is an example of CG56/IDF stars. This star receives one token, and sends the token out repeatedly. The repeat count depends on set port value. If the set port value is N, the input token is sent to output N times. The input and set ports receives a token respectively at the beginning of repeat. The control port is used only to determine a star schedule by the IDF (BDF) scheduler. At run time, tokens from the control port are not necessary, and are discarded.

(7)IRepeatE

This star is an example of CG56/IDF stars. This star receives tokens repeatedly from the input port, sums them, and sends the result out from the output port. The repeat count depends on the value of the set port. If the set port value is N, N input tokens are received, and one result is sent to the output port. The set ports receives a token at the beginning of repeat. The output port sends a token out at the end
of the repeat. The control port is used only to determine star schedule by the IDF (BDF) scheduler. In the execution, tokens from the control port are discarded.

The IRepeatB and IRepeatE stars can form a repeat process, in which each token produced by IRepeatB is processed by functional stars on the datapath between IRepeatB and IRepeatE stars. The IRepeatE star collects the processed tokens.

(8)IRepeat

This star realizes repeat process by only one star.

4.2.3. CG56/IDF demos

(1)ifthenelse

The ifthenelse demo is equivalent to the BDF ifThenElse demo. Difference between the both is that the Switch and Select stars are replaced with the ISwitch and ISelect stars. Results of the both are the same.

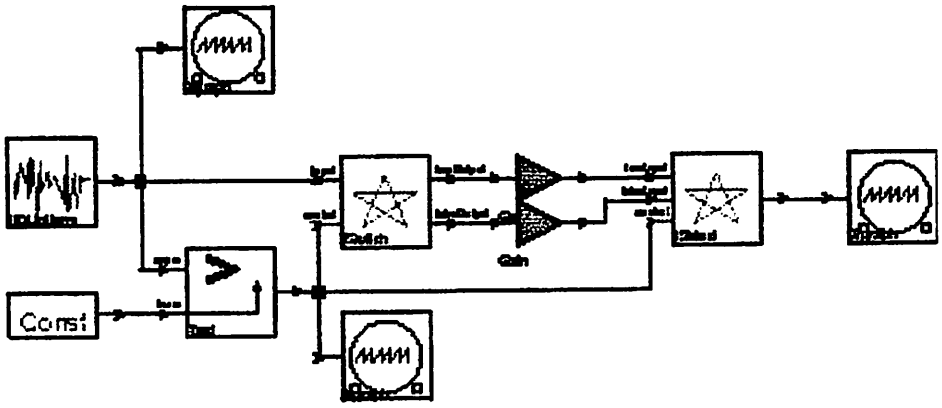

Fig. 17 ifThenElse demo in CG56/IDF

(2)Case4

The case4 shows an example of a case structure in program graphs. Source data from the IIDUniform star are switched to one of four Gain stars. The switching depends on the value of the control line of the ICaseB4 and ICAseE4 stars. The ICaseE4 star selects data through a gain. Two graphs show input and output data.

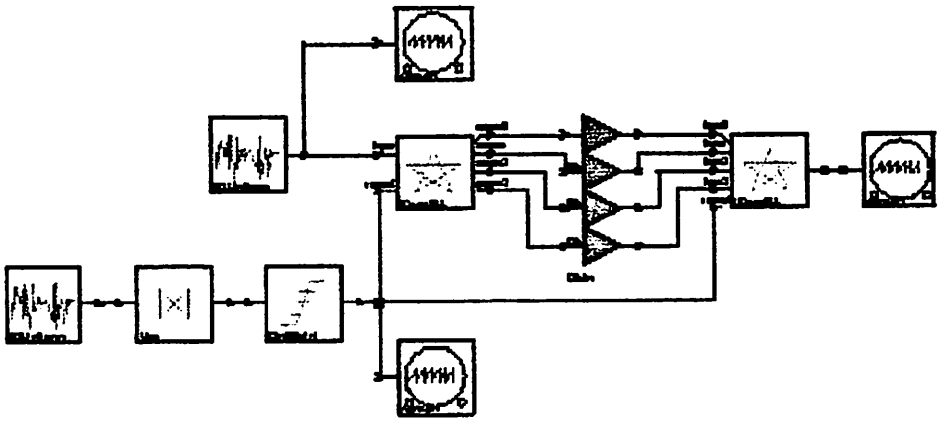

Fig. 18 Case4 demo in CG56/IDF

(3)Loop

The Loop demo shows an example of a loop structure in program graphs. The ILoop star read a source data from the IIDUniform star at the beginning of a loop process. At the same time, a loop count is also read from the set port. The data go round on the loop path between the loopFor and loopBack ports. At the end of loop, the looped data is put out from the output port. Two graphs show input and output data to/from the loop. CntlInI, cntlInO and cntlOut ports are used only to determine star schedule by the IDF (BDF) scheduler. Its connection must be the same as a loop structure demonstration of ILoop-demo. This restriction comes from the BDF scheduler, which is an original of the IDF scheduler. At run time, tokens from the control port are discarded.

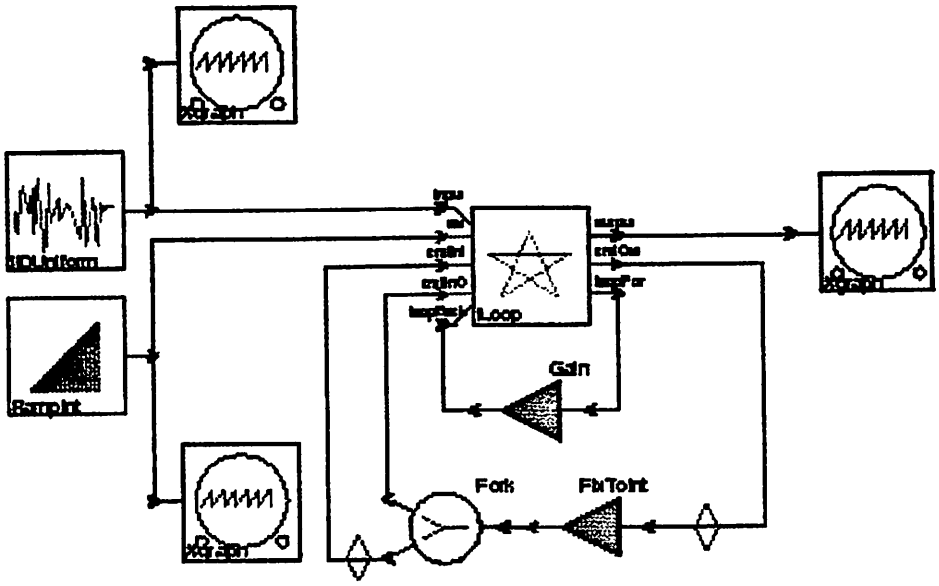

Fig. 19 Loop demo in CG56/IDF

(4)repeat

The repeat shows an example of a repeat structure in program graphs. The IRepeat star receives one data token from its input port, and a repeat count from its set port. The star sends the data tokens to beginRep port repeatedly. IRepeat star sums the tokens from endRep port. Two graphs show input and output data to/from the repeat. The paths to control ports of the IRepeat star are necessary, because of restriction of the BDF (IDF) scheduler. At run time, tokens on the control path are discarded.

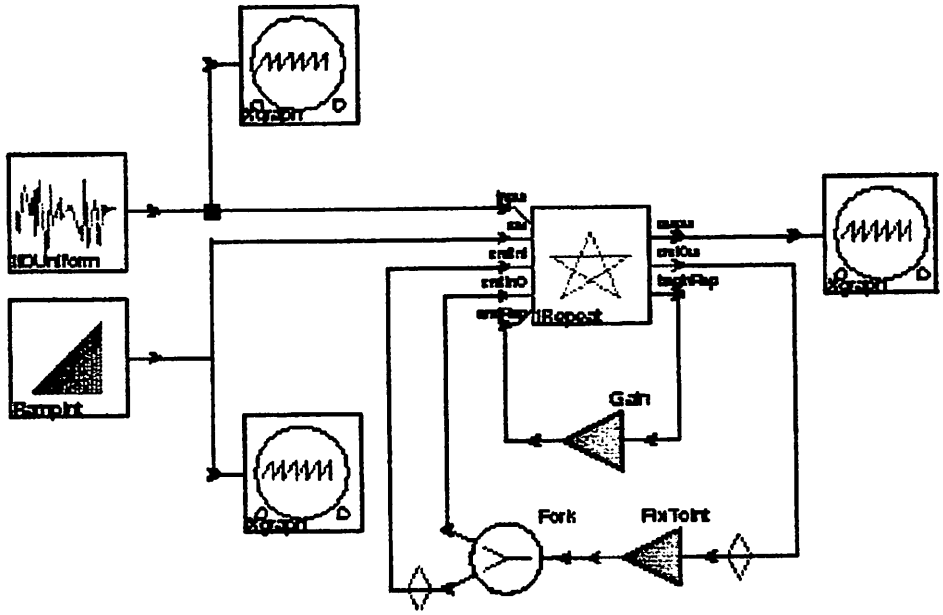

Fig. 20 Repeat demo in CG56/IDF

4.2.4. Writing CG56/IDF stars

Description of CG56/IDF stars also obeys to rules of writing CG56 stars. Additional descriptions are required to let the IDF (BDF) scheduler know dataflow control staffs and to control dataflow at run time.

For convenience, the ICaseB4 star is shown as an example of writing CG56/IDF stars.

```
name    { ICaseB4 }
domain  { CG56 }
```

The definitions of the star name and domain are common among all stars. The domain name is "CG56".

```
input
{
  name  { control }
  type  { int }
  desc  { control to select one out of conditional outputs }
}
```

Definition of input and output ports is also common among all stars. The star writer must be careful that the control port, whose value is used to determine behavior of a conditional port, must be a type of "int".

```
defstate
```

```
{
  name        { statOutput0 }
  type        { int }
  default     { 0 }
  attributes  { A_NONCONSTANT | A_YMEM | A_RAM }
  desc        { state variable of output0 }
}
```

State of conditional I/O ports, which are also used to determine behavior of a conditional port, should be defined in the defstate method. Its type is also a type of "int". Its values would be changed, so its attribute is usually "A_NONCONSTANT". The important thing is to assign a memory space to this state, so set the arribute to "A_YMEM | A_RAM", for example.

```
hinclude { "IdfCntl.h" }
```

Including a header file, "IdfCntl.h", is currently mandatory, except for BDF-like stars such as the ISwitch and ISelect stars.

```
protected
{
  IdfCntl cStatInput0;
  ......
}
```

The IdfCntl class is defined in "IdfCntl.h". Each conditional port needs IdfCntl class data to let the IDF scheduler know dataflow control staffs of the conditional port through the IDFPortHole::Set() method. The details will be described later.

```
method
{
  name { notone }
  type { int }
  arglist { "(CG56PortHole& port)" }
  code
  {
    return (port.numInitDelays() > 1 || port.far()->numXfer() > 1);
  }
}
```

This method is used to check the number of particle consumed or produced at portholes, and return FALSE (0) if not one. This is because porthole can consume or produce only one particle when it fires. This is the same as CGC/BDF stars. This method will appear in a setup method of the star, and be necessary to avoid unexpected result.

```
setup
```

```
{
  if(notone(input)    || notone(control) ||
    notone(output0) || notone(output1) ||
    notone(output2) || notone(output3))
  {
    Error::abortRun(*this,
        "Non-unity buffers connected to a switch not yet supported");
  }
```

(CONTINUED)

In the setup method of the star, function of notone() appears, first. This function is described above. It is wise to stop run, if the result of this function is FALSE (0), or unexpected result would worry you.

```
  else
  {
    // set control/status functions
    cStatOutput0.Set((const char*)fullName(),
          "control",
          "statOutput0",
          (const char*)evalFuncSymbol("ICase","SUB_CntlEq0"),
          (EVALFUNC)DUMMY_EVALFUNC,
          &statOutput0);
```

CntlState::Set() method stores dataflow control staffs in its object.

```
void CntlState::Set(const char* starName,
          const char* portName,
          const char* stateName,
          const char* funcName,
          EVALFUNC    efunc,
          IntState*   state)
```

starName ... this star name
portName ... control port name
stateName ... port state name
funcName ... evaluation function name
efunc ... pointer to an evaluatin function. It is OK to write "DUMMY_EVALFUNC".
state ... state variable

```
    // set relation
    output0.setRelation(DF_TRUE, &control, &cStatInput0);
```

PortHole::setRelation() method sets the relation of this port with associated porthole and dataflow control staffs.

void PortHole::setRelation(DFRelation relation, DynDFPortHole* assoc, CntlState* stat)

relation ... Specify the relation of this port with the result of the evaluation function.
DF_TRUE : produce/consume data only when the result of evaluation function is TRUE.
DF_FALSE : produce/consume data only when the result of evaluation function is FALSE.

```
initCode
{
 // add code of evaluation functions to the code stream
  addProcedure(COD_CntlEq0,"$sharedSymbol(ICase,SUB_CntlEq0)");

}
```

In initCode method, the code block of the evaluation function must be added in the procedure code section.

```
go
{
  // If embed() method is available, 'copy' codeblock is not necessary.
  addCode(copy);
}
```

In the go method, behavior of the star is described.

```
codeblock(copy)
{

}
```

Obey how to write CG56 codeblock.

```
codeblock(COD_CntlEq0)
{
        org     y:
$label(CData)
        dc      1                       ; constant data : 1
        org     p:
$sharedSymbol(ICase,SUB_CntlEq1)
        move    y:$label(CData),x1              ; load 1
        cmp     x1,a                    ; compare
        rts                             ; return
}
```

'COD_CntlEq0' codeblock is a description of an evaluation function. The function is user-definable. The rule is (1)The evaluation function must be a subroutine, because this instruction block is called as a subroutine. (2)When it returns, the condition code Z must reflect the result of the evaluation, ie, Z=1 and Z=0 mean TRUE and FALSE, respectively.

5. CONCLUSION

In this report, integer-controlled dataflow (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 static and conditional, so that memory requirement is determined at compile-time. 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.

ACKNOWLEDGEMENT

The author is grateful to Prof. Edward A. Lee of U.C. Berkeley, and to Brian Evans currently of University of Texas at Austin and Jose Luis Pino currently of HP.

REFERENCE

[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.