Software Synthesis from Dataflow Graphs

Shuvra S. Bhattacharyya

with contributions from William Plishker, Chung-Ching Shen and Hsiang-Huang Wu

Maryland DSPCAD Research Group
http://www.ece.umd.edu/DSPCAD/home/dspcad.htm

Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies,
University of Maryland, College Park, MD

Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011

Introduction

Applications and Tasks [Bhattacharyya 2011A]

Image: medical, computer vision, feature detection, etc.

Imaging device → Data preprocessing → Image reconstruction → Post reconstruction → Advanced image analysis → Image visualization

Video: coding, compression, etc.

→ Color processing → Prediction → Transformation & Quantization → Entropy Coding →

Audio: sample rate conversion, speech, etc.

Audio device → Data preprocesing → Feature extraction → Data postprocessing

Wireless communication systems

→ Source encoding → Channel encoding → Digital modulation → D/A conversion → RF Back-end

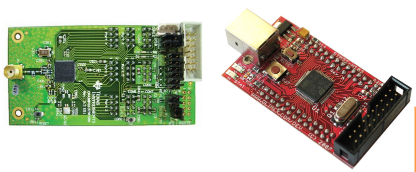
Platforms

Programmable DSP

GPU

FPGA

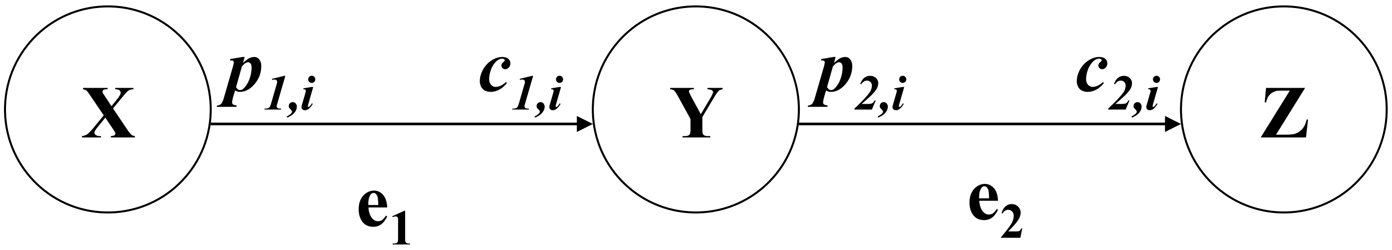
Microcontroller



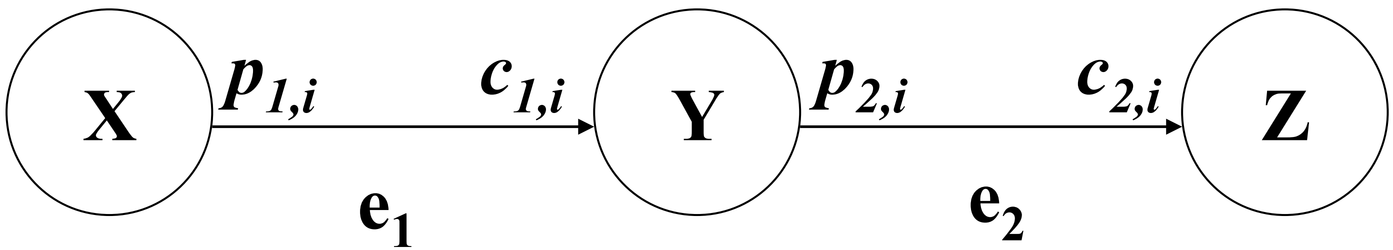
S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

Background: Dataflow Graphs

- Vertices (actors) represent computation
- Edges represent FIFO buffers
- Edges may have delays, implemented as initial tokens
- Tokens are produced and consumed on edges
- Different models have different rules for production and consumption (SDF → fixed, CSDF → periodic, BDF → dynamic in terms of "control tokens")



Dataflow Production and Consumption Rates



- $p_{x,y}$ denotes the number of tokens produced onto edge e_x by the yth firing of its source actor (for $y = 1, 2, \ldots$).
- Similarly, $c_{x,y}$ denotes the number of tokens consumed from edge e_x by the yth firing of its sink actor.

S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*.
Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

Dataflow Models of Computation

- Used widely in design tools for DSP
- Application is modeled as a directed graph
  - Nodes (actors) represent functions
  - Edges represent communication channels between functions
  - Nodes produce and consume data from edges
  - Edges buffer data in a FIFO (first-in, first-out) fashion
- Data-driven execution model
  - An actor can execute whenever it has sufficient data on its input edges
  - The order in which actors execute is not part of the specification
  - The order is typically determined by the compiler, the hardware, or both
- Iterative execution
  - Body of a loop to be iterated a large or infinite number of times

S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

DSP-oriented Dataflow models: State of the Art

- A variety of useful dataflow models with important trade-offs involving flexibility, and supported methods for analysis and optimization
- Increasingly used for simulation and rapid prototyping, using automated techniques for scheduling, memory management, and other key design flow processes
- Diverse target platforms → "Implementation gap"

Some Useful Dataflow Models

Model	Abbr.	Rule	Stc/Dyn
Synchronous Dataflow	SDF	Fixed firing behavior for all actors	Static
Cyclo-static Dataflow	CSDF	Periodic firing behavior	Static
Boolean Dataflow	BDF	Firing behavior may be contingent on the value of a Boolean token	Dynamic
Parameterized Synchronous Dataflow	PSDF	Firing behavior may be changed between iterations of the application graph	Dynamic
Enable Invoke Dataflow	EIDF	Modes have fixed behavior, but actors may dynamically switch between modes	Dynamic
Core Functional Dataflow	CFDF	A deterministic subclass of EIDF in which the next mode of an actor is always unique	Dynamic

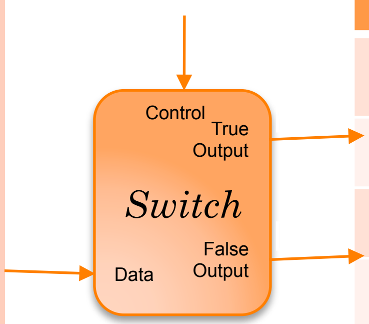
Cross Platform Design and Scheduling

- Lightweight dataflow [Shen 2010, Shen 2011]
  - A dataflow programming approach for model-based design and implementation of DSP systems.
  - Easily retargetable across platform-oriented languages for actor implementation.
  - Lightweight dataflow "glue" as an attractive target for software synthesis.
- The dataflow schedule graph [Wu 2011]
  - A formal model for representing and manipulating dataflow graph schedules that is *itself* dataflow-based.
  - Accommodates a wide variety of dataflow application models and target architectures.
  - A novel framework for designing and specifying schedules.

Core Functional Dataflow (CFDF)

- Divide actors into sets of *modes* [Plishker 2008]
  - Each mode has a fixed consumption and production behavior, but actors may dynamically switch between modes.
- Write the enabling conditions for each mode
- Write the computation associated with each mode
  - Including determining *next mode* to enable and then invoke
- For example, consider a standard Switch actor:

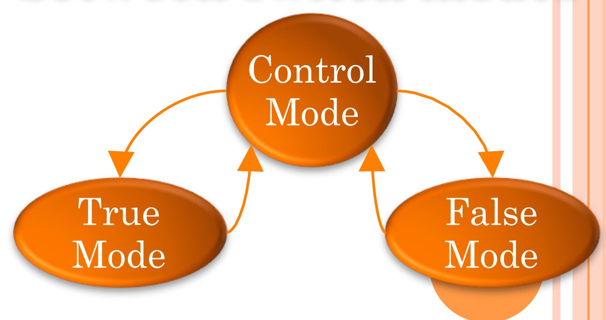
Switch Actor



Production & consumption behavior of switch modes

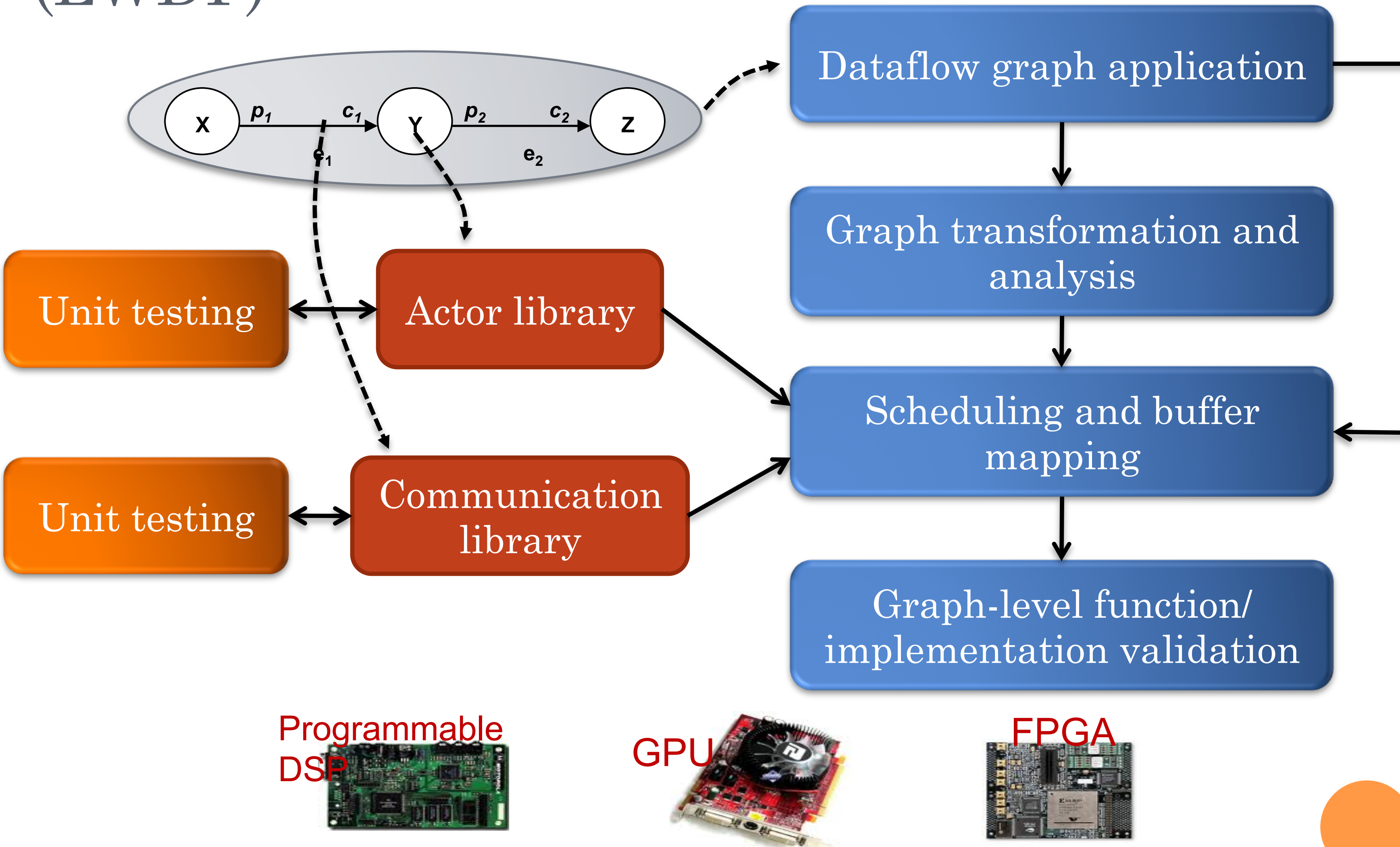
Mode	Consumes		Produces	
	Control	Data	True	False
Control	1	0	0	0
True	0	1	1	0
False	0	1	0	1

Mode transition diagram between switch modes



Design Flow using Lightweight Dataflow (LWDF)

[Shen 2010]



S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

Lightweight Dataflow Programming Approach

[Shen 2010]
[Shen 2011]

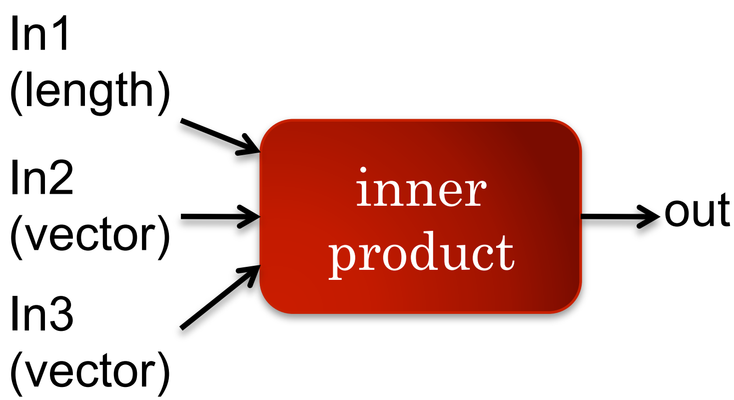
- A dataflow programming approach for model-based design and implementation of DSP systems.
  - "lightweight"→ minimally intrusive on existing design processes, and requires minimal dependence on specialized tools or libraries.
- Features
  - Improve the **productivity** of the design process and the quality of derived implementations.
  - **Retargetability** across different platforms.
  - Allow designers to **integrate and experiment with dataflow modeling approaches** relatively quickly and flexibly within existing design methodologies and processes.

LWDF Design Principles

- Each actor has an **operational context (OC)**, which encapsulates
  - parameters
  - mode variables
  - local variables
  - references to the FIFOs corresponding to the input and output ports of an actor as a component of the enclosing dataflow graph.
  - references to the "execution functions" of an actor.
    - Enable function
    - Invoke function

S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

Operational Context – Example in C



- This actor involves a sequence p1, p2, …, pN of "processing modes", where the dataflow rates vary as a function of the mode index
- Such a sequence can be implemented through common (parameterized) control

```
typedef struct {

    /* local variables */
    int length;

    /* mode variable */
    int mode;

    /* references to the FIFOs */
    fifo_pointer in1;
    fifo_pointer in2;
    fifo_pointer in3;
    fifo_pointer out;

    /* references to the
       execution functions */
    actor_enable_function_type enable;
    actor_invoke_function_type invoke;

} inner_product_context_type;
```

LWDF Design Principles

- Methods that are involved in the implementation of an actor
  - ***Construct***: connects an actor to its input and output edges (FIFO channels), and performs any other pre-execution initialization associated with the actor.
  - ***Enable and Invoke***: implement the CFDF semantics associated with an actor firing.
  - ***Terminate***: performs any operations that are required for "closing out"' the actor after the enclosing graph has finished executing

S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

Methods – Examples in C


```
inner_product_co
    in1, fifo_po
{
    inner_produc
    context = ut
    context->ena
    context->inv
    context->mod
    context->len
    context->in1
    context->in2
    context->in3
    context->out
    return conte
}
```

```
void inner_product_invoke(inner_product_context_type *context) {
    int i = 0;
    int sum = 0;
    int x_value = 0;
    int y_value = 0;
    switch (context->mode) {
        case STORE_LENGTH:
            fifo_read(context->in1, &context->length);
            if (context->length < 0) {
                context->mode = INNER_PRODUCT_ERROR_LENGTH;
                return;
            }
            context->mode = PROCESS;
            break;
        case PROCESS:
            sum = 0;
            for (i = 0; i < context->length; i++) {
                fifo_read(context->in1, &(x_value));
                fifo_read(context->in2, &(y_value));
                sum += (x_value * y_value);
            }
            fifo_write(context->out, &sum);
            context->mode = STORE_LENGTH;
            break;
        default:
            context->mode = INNER_PRODUCT_ERROR_INVALID;
            break;
    }
    return;
}
```

```
*context) {

ontext->length) &
t->length);
```

LWDF-C Actor APIs

Type Definitions:

```
/* An actor's operational context (OC). */
typedef struct {
    /* parameters */
    /* local and mode variables */
    /* references to FIFO pointers */
    /* reference to a pointer of actor's execution functions */
} [actor_name]_context_struct;

/* A pointer to actor enable/invoke functions, which are functions that
    executes an actor with a given context. */
typedef void (*actor_[enable/invoke]_function_type) (struct
    actor_context_struct *context);
```

Key Methods:

```
[actor_name]_context_type *[actor_name]_new(…);
void [actor_name]_enable([actor_name]_context_type *context);
void [actor_name]_invoke([actor_name]_context_type *context);
void [actor_name]_terminate([actor_name]_context_type *context);
```

LWDF-C FIFO APIs

Type Definitions:

```
/* A FIFO. */
typedef struct _fifo_struct fifo_type;
/* A pointer to a fifo. */
typedef fifo_type *fifo_pointer;
```

Key Methods:

```
fifo_pointer fifo_new(int capacity, int token_size);
int fifo_population(fifo_pointer fifo);
int fifo_capacity(fifo_pointer fifo);
void fifo_write(fifo_pointer fifo, void *data);
void fifo_write_block(fifo_pointer fifo, void *data, int size);
void fifo_read(fifo_pointer fifo, void *data);
void fifo_read_block(fifo_pointer fifo, void *data, int size);
```

SUMMARY: ACTOR DESIGN TEMPLATE

- **Construct** function
  - lide_c_<*actor name*>_context_type *lide_c_<*actor name*>_new (<*FIFO pointer list*>, *[parameter list]*);
- **Enable** function
  - boolean lide_c_<*actor name*>_enable (lide_c_<*actor name*>_context_type *context);
- **Invoke** function
  - void lide_c_<*actor name*>_invoke (lide_c_<*actor name*>_context_type *context);
- **Terminate** function
  - void lide_c_<*actor name*>_terminate (lide_c_<*actor name*>_context_type *context);

S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

Dataflow-based Software Development

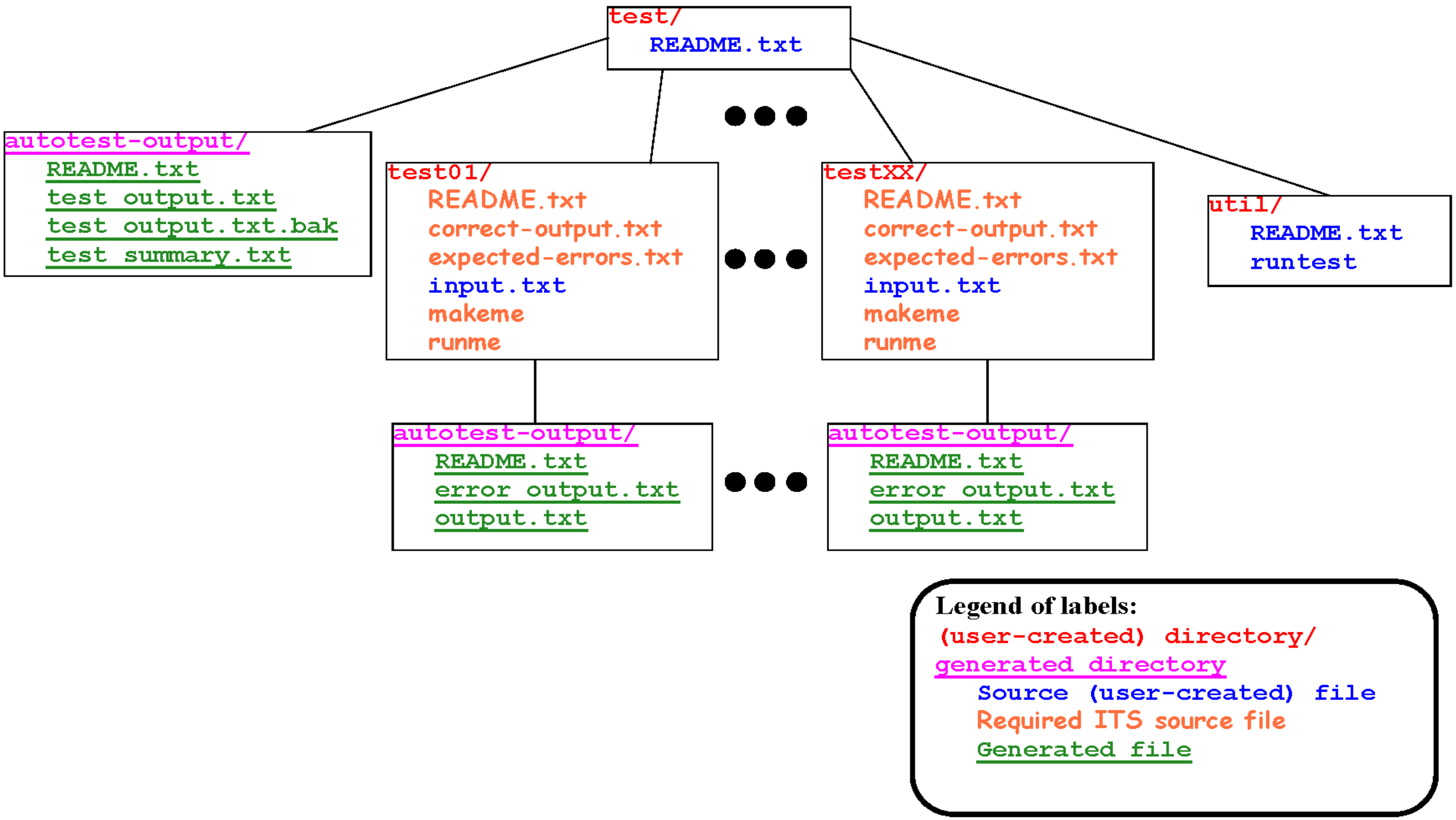
- The Lightweight Dataflow Design Environment (LIDE)
  - An easy-to-install, -learn, and -maintain software development environment for guiding designers in experimenting with dataflow-based approaches for design, implementation, and testing of digital signal processing systems.
  - A Bash-based environment, supported on Linux, MacOS, and Windows/Cygwin.
  - Contains a collection of dataflow graph elements (actors and FIFOs) as design templates for building signal processing systems that are specified using dataflow concepts.
  - Based on the semantics of a particular dataflow model called *Enable-Invoke Dataflow* [Plishker 2008], as well as a lightweight dataflow (LWDF) programming method [Shen 2010].
  - Integrated with the DICE unit testing framework [Bhattacharyya 2011B].
    - A cross-platform, language-agnostic engine for automated testing

Synergy / Integrated Use with DICE

- For high productivity, high reliability hardware/software development, testing should be integrated deeply into the design process.
- Providing effective tests is an important part of overall project design.
- Modularity and data streaming characteristics of dataflow actors facilitate efficient testing.
- Automated test execution is critical to practical use of testing methods.
- DICE as a cross-platform, language-agnostic engine for automated testing.

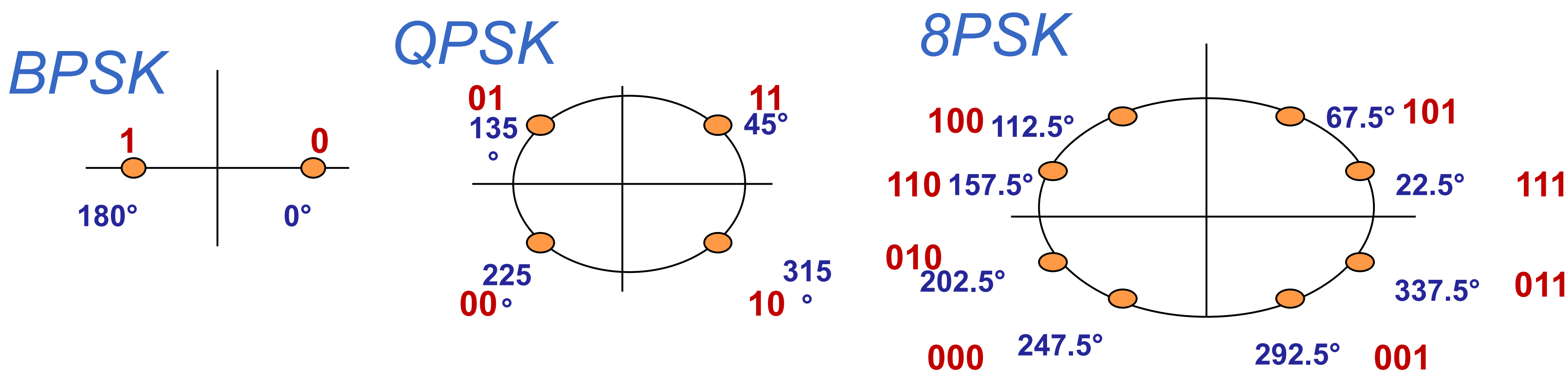
S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

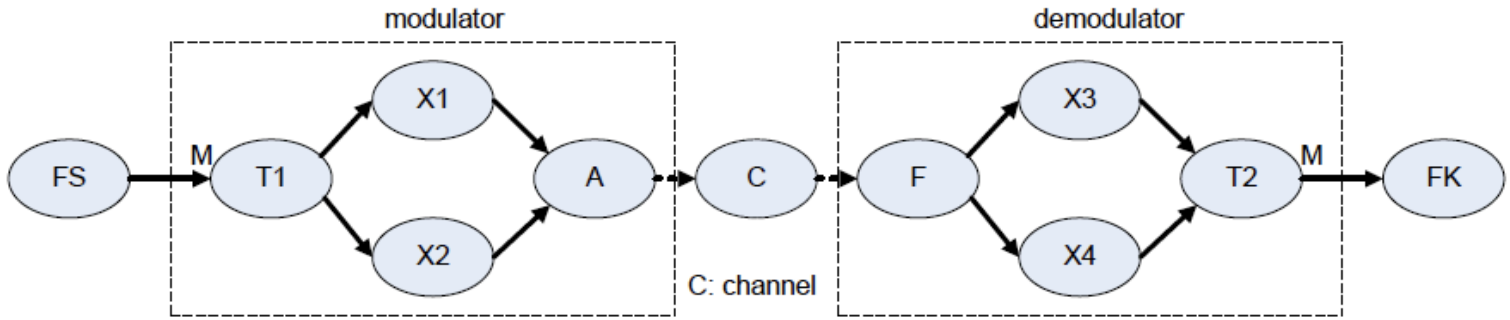
Test Subtree Based on DICE



S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

Design Example: Reconfigurable Phase-Shift Keying





FS: binary bits generator
T1: table look-up for phase
X1: multiplier for Acost
X2: multiplier for Asint
A: adder

F: forked and duplicated modulated signal
X3: multiplier for Acost
X4: multiplier for Asint
T2: table look-up for binary bits
FK: binary bits presenter

- M=1 for BPSK, M=2 for QPSK, and M=3 for 8PSK

S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

Design Example: Reconfigurable Phase-Shift Keying

Application simulation driven by a simple scheduler

```
void util_simple_scheduler(actor_context_type *actors[], int actor_count,
        char *descriptors[]) {
        boolean progress = FALSE;
         int i = 0;
        do {
                progress = 0;
                for (i = 0; i < actor_count; i++) {
                        progress |= util_guarded_execution(actors[i], descriptors[i]);
                }
        } while (progress);
}
```

guarded execution for CFDF model

- Simulation based on LWDF-C
  - 3GHz Intel Pentium PC with 2GB of RAM
  - Input bit stream: 10,000 bits
  - Simulation time: 1.5 seconds

Retargeting to a hardware implementation

- Application execution driven by self-timed scheduling strategy
  - That is, an actor module fires whenever it has sufficient tokens available on its input FIFOs
- FPGA implementation based on LWDF-V (integration of LWDF environment and Verilog for actor programming)
  - Target FPGA device: Xilinx Virtex-4
  - Resource utilization after synthesis: 1,484 LUTs (5% util. rate) and 1,464 CLBs (10% util. rate)

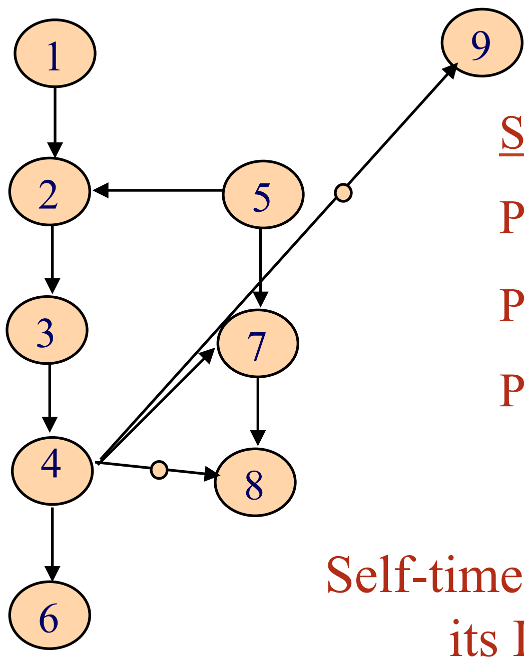
CROSS PLATFORM DESIGN AND SCHEDULING

- **Lightweight dataflow** [Shen 2010, Shen 2011]
  - A dataflow programming approach for model-based design and implementation of DSP systems.
  - Easily retargetable across platform-oriented languages for actor implementation.
  - Lightweight dataflow "glue" as an attractive target for software synthesis.
- **The dataflow schedule graph** [Wu 2011]
  - A formal model for representing and manipulating dataflow graph schedules that is *itself* dataflow-based.
  - Accommodates a wide variety of dataflow application models and target architectures.
  - A novel framework for designing and specifying schedules.

Dataflow Graph Scheduling

- Assigning actors to processors, and ordering actor subsets that share common processors
- Here, a "processor" means a hardware resource for actor execution on which assigned actors are time-multiplexed
- Communication can be modeled with send and receive actors
- Scheduling objectives include
  - Exploiting parallelism
  - Buffer management
  - Minimizing power/energy consumption

Schedule and Schedule Modeling Example



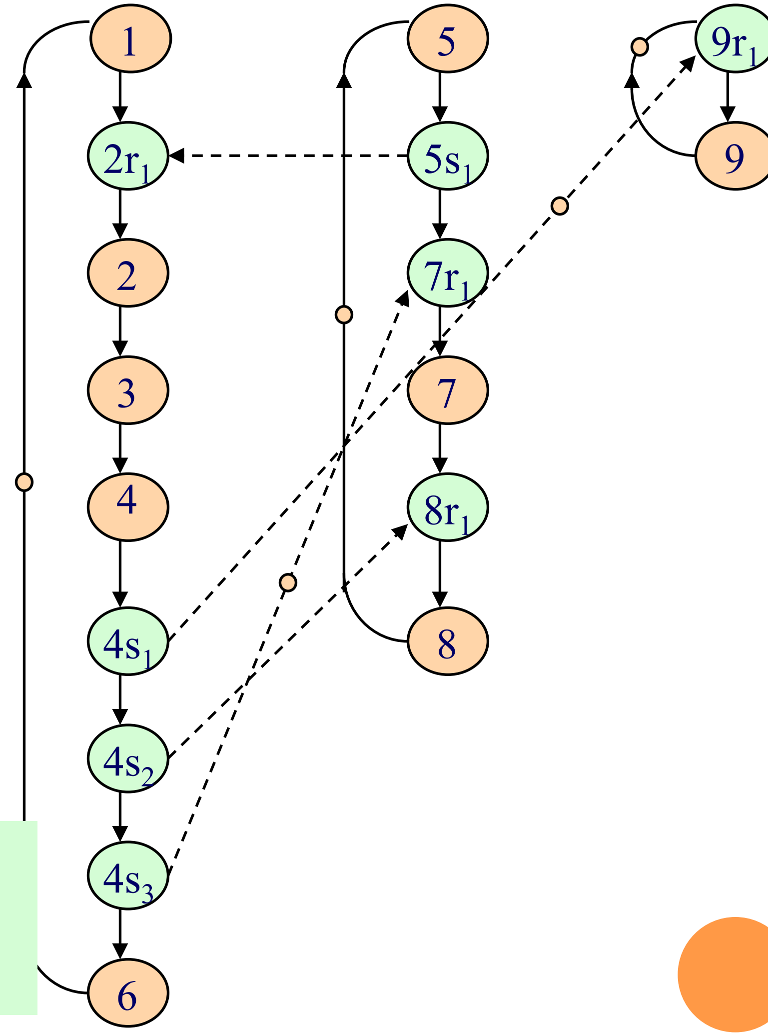
Self-Timed Schedule

Proc 1: (1, 2, 3, 4, 6)

Proc 2: (5, 7, 8)

Proc 3: (9)

Self-timed schedule and its IPC graph



Every edge (x, y) induces the following precedence constraint:
$start(y, \mathrm{k}) \geq start(x, k - delay((x, y)) + t(x)$

S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

The Dataflow Schedule Graph: Objectives

- Provide a common representation for representing an working with dataflow graph schedules
- Flexible support for alternative application-level dataflow models
- Modeling of both sequential and parallel schedules
- Unified modeling for static, dynamic, and quasi-static schedules
- Interoperability with other schedule models

S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

Previous Dataflow Graph Schedule Representations

- SDF firing sequences
  - Buffer minimization
- Time tables (fully static schedules)
- Static looped schedules
- Parameterized looped schedules
- Schedule trees
- Interprocessor communication graphs and synchronization graphs (self-timed schedules)

DSG: Distinguishing Characteristics

- A formal, schedule model that is simultaneously:
  - dataflow based;
  - capable of handling dynamic schedule structures;
  - capable of handling dynamic dataflow application models.
- The underlying actor model is EIDF/CFDF
  - General dynamic dataflow model.
  - Allows specialized models to be extracted systematcially through mode analysis.

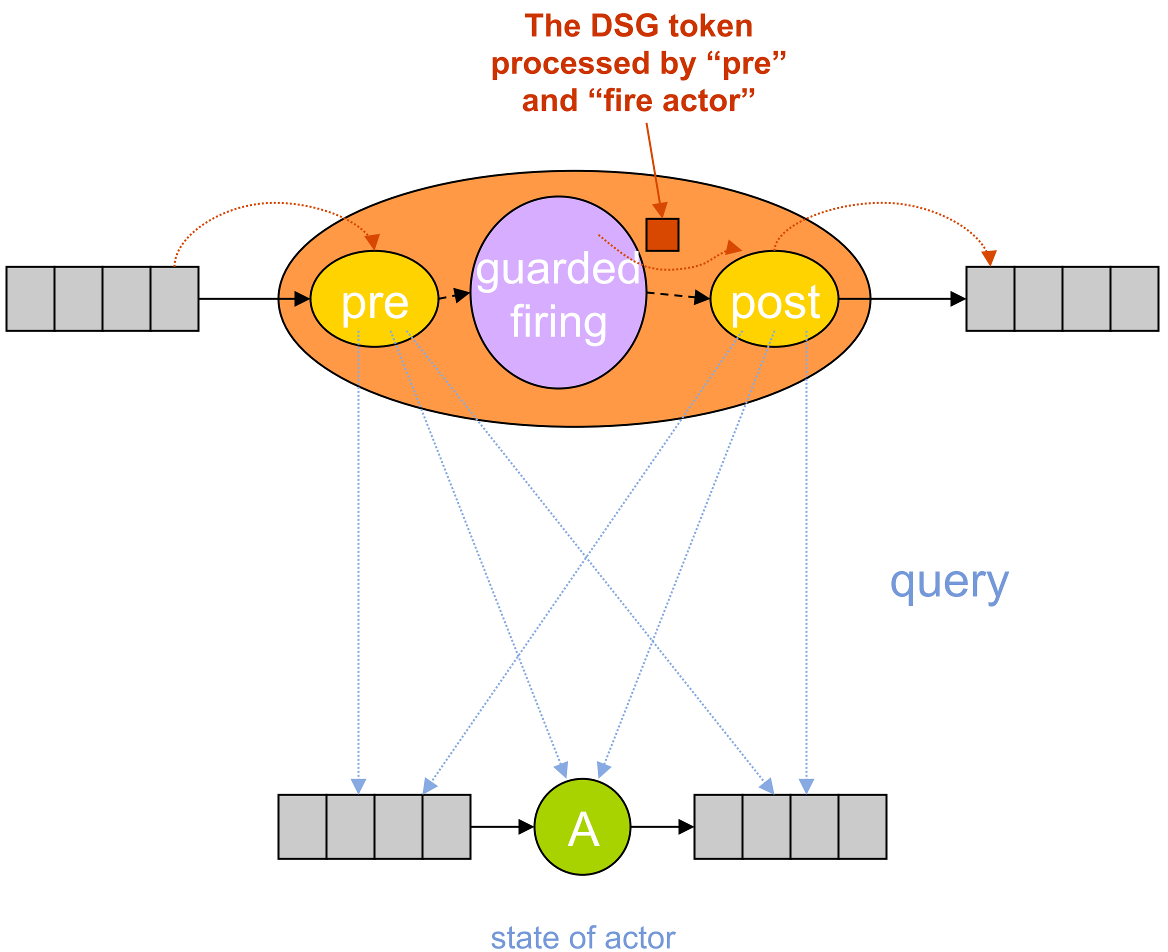
DSG Tokens

- The DSG does not explicitly model data communication between actors → this is already modeled in the application-level dataflow graph.
- Tokens that flow along DSG edges serve to enable actors for execution
  - Like an actor-level "program counter" for the target processor.
- DSG tokens can also contain values that are manipulated or queried to achieve customizable forms of schedule control.

Reference Actors

- An RA can be viewed as a "pointer to" or "wrapper for" an application graph actor.
- Single input, single output.
- Homogeneous SDF (HSDF).
- Intuitively, an RA specifies the computation that is executed when the corresponding application graph actor is "visited" during schedule execution.

Internal Structure of an RA



S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*.
Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

Firing an RA

Firing of an RA involves the following sequence of steps:

1) The RA consumes a token from its input edge. This token is passed as input to pre_A, which executes, and updates the state of RA.
2) A guarded execution of ref_A is carried out. That is, ref_A is fired once if it is enabled.
3) An execution of $post_A$ is carried out. This execution operates on the state of the RA. The output value from this execution is produced as the output of the RA firing.

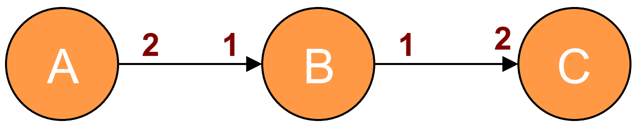
S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

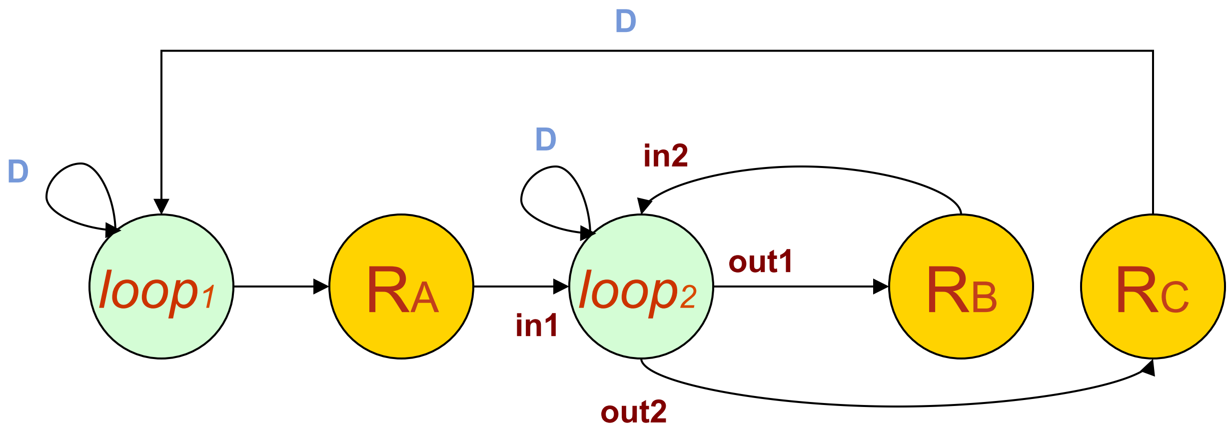
Schedule Control Actors

- Flexible modeling of dynamic scheduling structures
- Not restricted to pre-defined schemas
- Lumped homogeneous synchronous dataflow (LHSDF) model:
  total input port consumption = total output port consumption = 1
- Self-loops (state models) for SCAs "do not count" in the LHSDF restriction

Sequential DSG Example

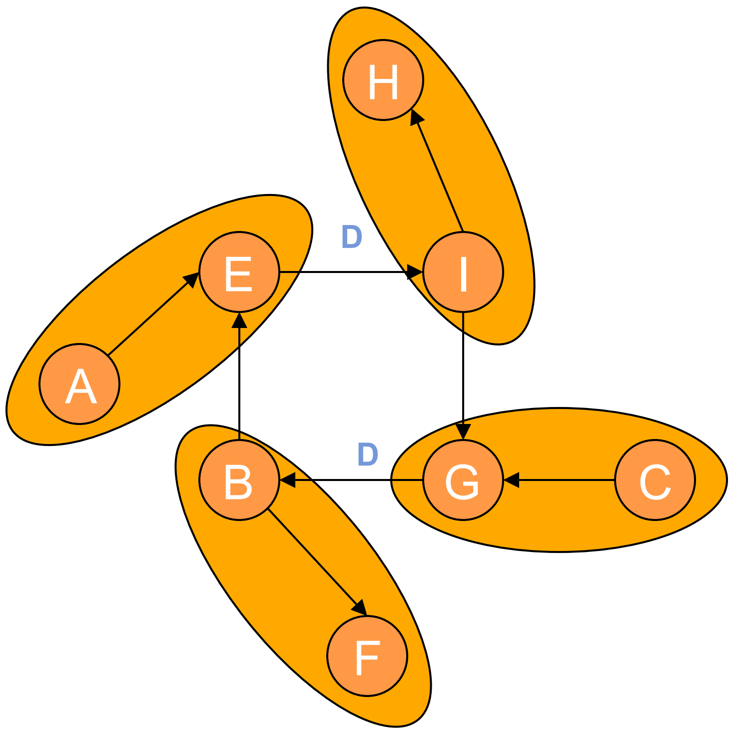
(A(2B)C)





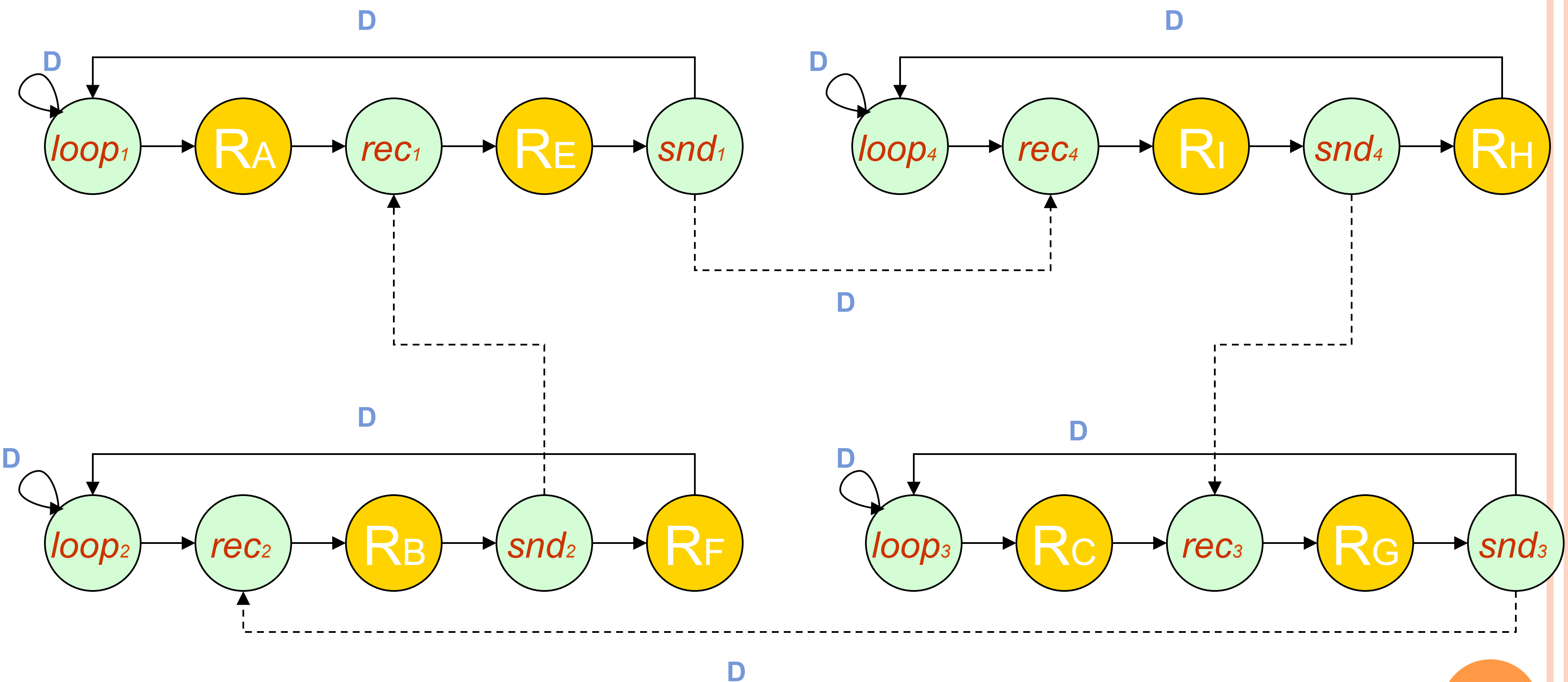
S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

Concurrent DSG Example: Application Graph and Partition onto Multiple Processors



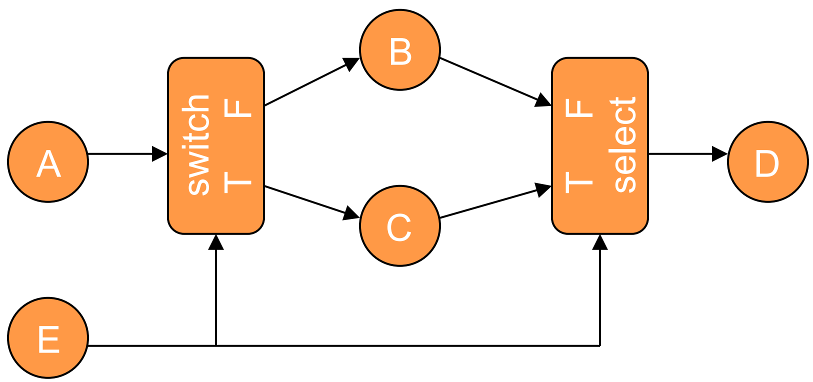
S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

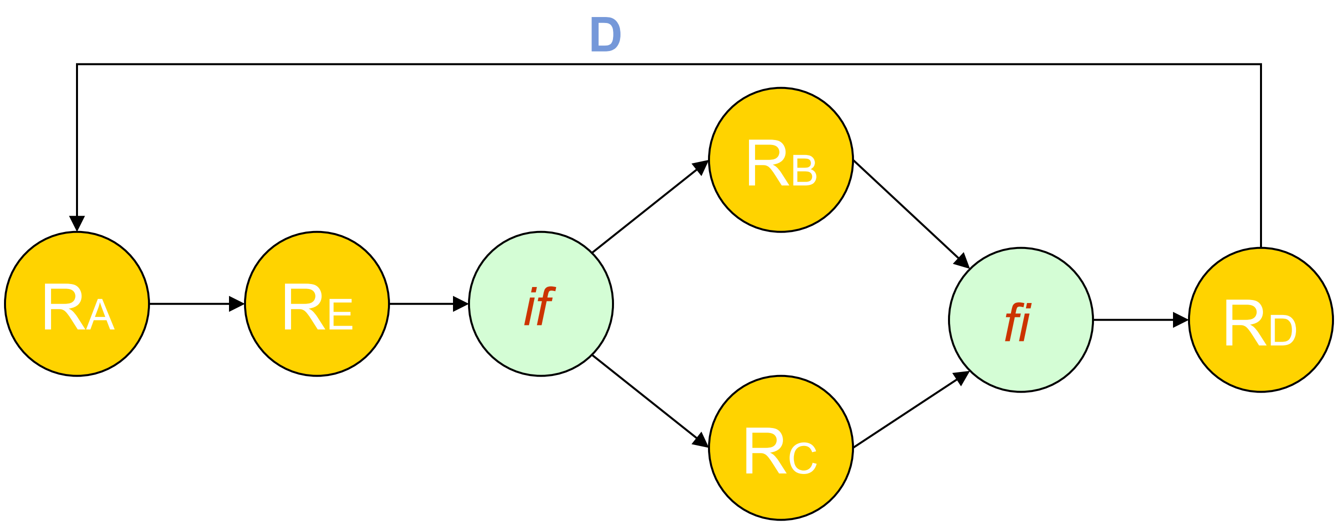
Concurrent DSG Example: Schedule Model



S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*.
Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

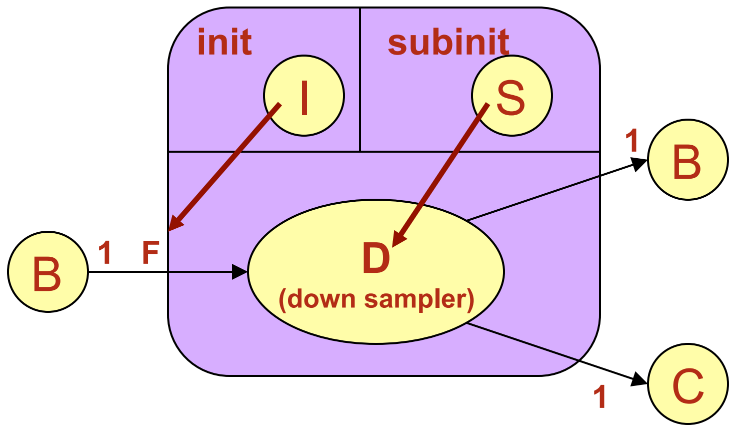
Adaptive DSGs: Sequential DSG Models for Dynamic and Quasi-Static Schedules



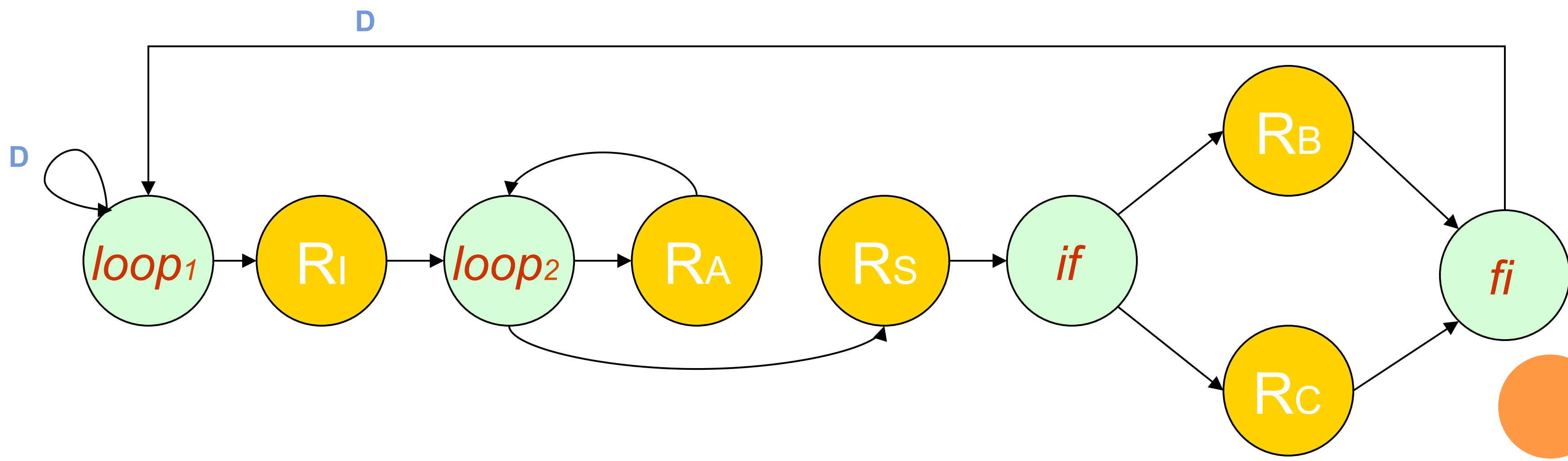


S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

Example: Parameterized Downsampler-Selector



- Parameterized dataflow application model
- Dynamic downsampling factor and output selection parameter (Boolean)

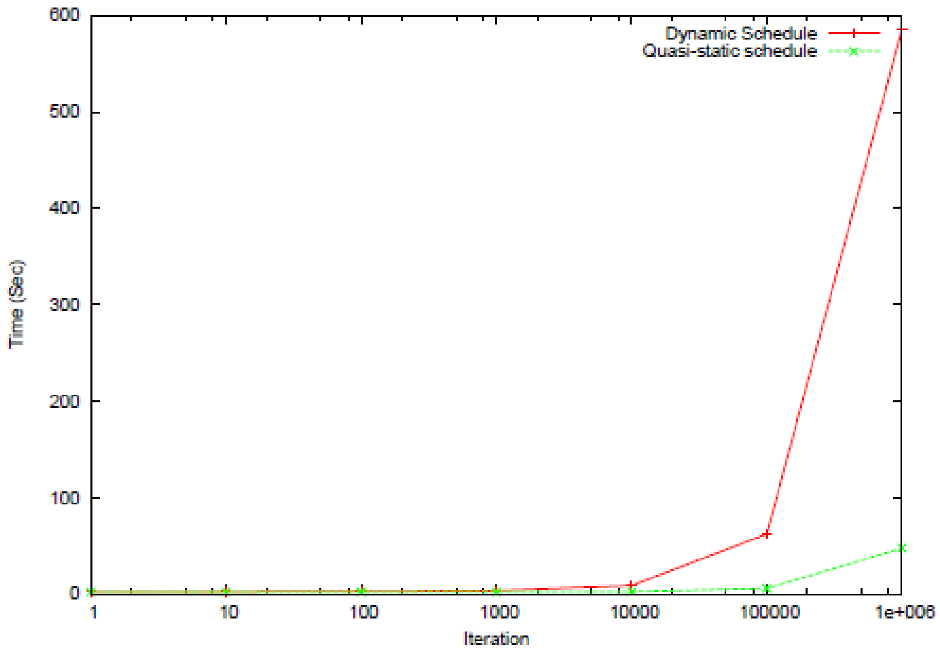


S. S. Bhattacharyya, W. Plishker, C. Shen, H. Wu, *Software Synthesis from Dataflow Graphs*. Presented at the Workshop on Software Synthesis, Taipei, Taiwan, October 14, 2011.

Experimenting with Alternative Schedules for a Downsampling Subsystem

- Dynamic schedule: Canonical CFDF schedule (round robin with guarded execution)
  - More general, easier to apply
  - Faster to construct/deploy
- Quasi-static schedule: an optimized parameterized looped schedule
  - Potential for improved performance and predictability
- The DSG representation captures these alternatives in a common, dataflow-based format for efficient incorporation into tools (e.g., PSDFSim) and design processes

Dynamic Schedule (Sec.)							
Iteration	1	10	10^2	10^3	10^4	10^5	10^6
CPU time	1.00	1.06	0.97	0.97	1.16	1.19	320.70
Total time	**2.29**	**2.35**	**2.60**	**3.21**	**8.65**	**62.08**	**585.97**
Quasi-static schedule (DSG) (Sec.)							
CPU time	0.60	0.59	0.59	0.59	0.59	0.61	0.64
Total time	**1.86**	**1.87**	**1.93**	**1.95**	**2.28**	**5.67**	**47.45**



Experimenting with Alternative Schedules for the Downsampling Subsystem

- Dynamic schedule (recompute the schedule every time graph parameters change)
  - More general, easier to apply
  - Faster to construct/deploy
- Quasi-static schedule: an optimized parameterized looped schedule
  - Potential for improved performance and predictability
- The DSG representation captures these alternatives in a common, dataflow-based format for efficient incorporation into tools (e.g., PSDFSim) and design processes

Summary

- Foundations for software synthesis from dataflow graphs are provided by a variety specialized dataflow models of computation.
- Widening implementation gap between the diversity of abstract models and concrete platforms.
- Cross platform implementation, validation, and scheduling methods
  - Lightweight dataflow
  - DICE
  - The dataflow schedule graph

Software Release News, Updates, and Announcements

- One can subscribe to the "DSPCAD News" email list by sending a request to

  dspcad-manager@listserv.umd.edu

- Includes announcements about LIDE, DICE, and other resources

Questions and Discussion

Acknowledgements

- This work was sponsored in part by the Laboratory for Telecommunication Sciences, US Air Force Research Laboratory, and US National Science Foundation.

References

- [Bhattacharyya 2011A] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors. *Handbook of Signal Processing Systems*. Springer, 2010.
- [Plishker 2008] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya. Functional DIF for rapid prototyping. In *Proceedings of the International Symposium on Rapid System Prototyping*, pages 17-23, Monterey, California, June 2008.
- [Shen 2010] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya. A lightweight dataflow approach for design and implementation of SDR systems. In *Proceedings of the Wireless Innovation Conference and Product Exposition*, pages 640-645, Washington DC, USA, November 2010.

For more details on these publications:
http://www.ece.umd.edu/DSPCAD/home/dspcad.htm

References

- [Shen 2011] C. Shen, W. Plishker, and S. S. Bhattacharyya. Dataflow-based design and implementation of image processing applications. Technical Report UMIACS-TR-2011-11, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011. http://drum.lib.umd.edu/handle/1903/11403.
- [Bhattacharyya 2011B] S. S. Bhattacharyya, W. Plishker, C. Shen, N. Sane, and G. Zaki. The DSPCAD integrative command line environment: Introduction to DICE version 1.1. Technical Report UMIACS-TR-2011-10, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011. http://drum.lib.umd.edu/handle/1903/11422.
- [Wu 2011] H. Wu, C. Shen, N. Sane, W. Plishker, and S. S. Bhattacharyya. A model-based schedule representation for heterogeneous mapping of dataflow graphs. In *Proceedings of the International Heterogeneity in Computing Workshop*, pages 66-77, Anchorage, Alaska, May 2011.

For more details on these publications:

http://www.ece.umd.edu/DSPCAD/home/dspcad.htm