# AMI Analysis Using a Proxy Class

DesignCon IBIS Summit
Santa Clara, CA
February 3rd, 2017

Wei-hsing Huang, SPISim
Wei-hsing.Huang@spisim.com

# Agenda:

- Background
- Motivation
- Approach: A proxy class
- Analysis/Experiments:
  - Stress and consistency tests
  - Co-optimization within internal process
  - Co-optimization/Simulation with external process
- Summary
- Q & A



SPISIM

# Background: Roles in an AMI flow

- AMI simulation flow: [1]
  - Simulation platform and AMI models
    - Communicate via AMI API defined in IBIS spec.
  - Exchange settings via “Reserved parameters”

- Back-channel co-optimization flow: [2][3][4]
  - Simulation platform, Protocols, and AMI models
  - Models exchange settings via “BCI*” parameters
  - Different types of models
    - Legacy, hardware protocol backed or others

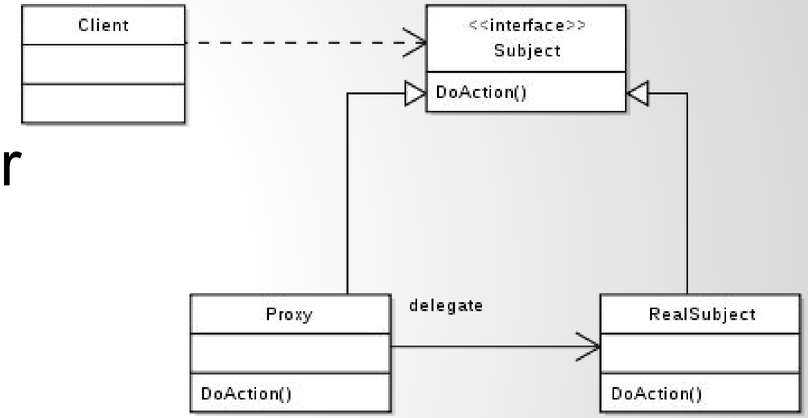
# Motivation: Bridge the gaps

- A “mediator” in the flow
  - Between simulator and models, or between different models
  - Mainly for development purpose (testing and experiments)

- For model & optimization development:
  - Don’t want to wait until BIRD 147.4/Spec. finalized
  - My favorite simulator may not support new spec. or get updated
  - Want to use existing simulator and models, now
  - Need to process data outside simulator and model
  - Need to reuse simulator’s post-process functions

# Approach: A Proxy Class

- A proxy class: [5]
  - Implements AMI API
  - Called/Loaded by simulator
  - Calls/loads actual models
  - Is a “Man in the middle” [6]
    - Can intercept, modify data
    - Can perform customized flow




Proxy Pattern UML

- A proxy class is an AMI-compatible .dll(s)/.so(s) which loads actual model’s AMI .dll(s)/.so(s) and does things...

# Proxy class code snippet:

```
//------------------------------- AMI PROXY --------------------------------
#include <windows.h>

/* Init: in statistical mode, processing impulse response. */
/*       in bit-by-bit  mode, initialize data structures.  */
typedef long (*amiInit)(double *htInput, long rowSize, long numAggr,
                        double sampInt, double bitTime, char *inpParm,
                        char **outParm, void **modlPtr, char **message);

/* GetWave: for bit-by-bit simulation */
typedef long (*amiGWav)(double *wavData, long wavSize, double *clkTime,
                        char **outParm, void *modlPtr);

/* Close: to clean-up allocated memory */
typedef long (*amiFini)(void *modlPtr);
//---------------------------- End of TypeDef ------------------------------

/* Proxy's Init funciton: called by the simulator and deligate to real model */
IBIS_AMI_API long AMI_Init(double *htInput, long rowSize, long numAggr,
                           double  sampInt, double  bitTime, char *inpParm,
                           char   **outParm, void  **modlMem, char **message) {
  HMODULE dllLibs = LoadLibrary("C:/Temp/SPISimAMI_Tx.dll");
  amiInit ptrInit = (amiInit) GetProcAddress(dllLibs, "AMI_Init");
  long initStatus = (ptrInit)(&htInput[0], rowSize, numAggr, sampInt, bitTime,
                              inpParm, outParm, modlMem, message);
  FreeLibrary(dllLibs);
  return initStatus;
}
```

# **Analysis 1:** Consistency and stress tests

- Waveform results of first and last loop should be the same
- Monitor resource, memory usage should stay roughly the same
- Important as AMI_GetWave may be called many times for lengthy bits

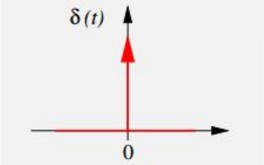
```
/* The following is proxy class's getWave function, called by the simulator */
IBIS_AMI_API long AMI_GetWave(double *wavData, long  wavSize, double *clkTime,
                              char  **outParm, void *modlPtr) {

  HMODULE dllLibs = LoadLibrary("C:/Temp/SPISimAMI_Tx.dll");
  amiGWav ptrGWav = (amiGWav) GetProcAddress(dllLibs, "AMI_GetWave");
  double *tstData = (double*)calloc(wavSize, sizeof(double));
  // loop for stress and consistency tests
  for (int i = 0; i < 10000; i++) {
    // duplicate test data from original
    memcpy(tstData, wavData, sizeof(double) * wavSize);
    // call model's getWave function
    assert((ptrGWav)(tstData, wavSize, clkTime, outParm, modlPtr) == 1);
  }
  FreeLibrary(dllLibs);
  free(tstData);
  tstData = 0;
  return 1;
}
```

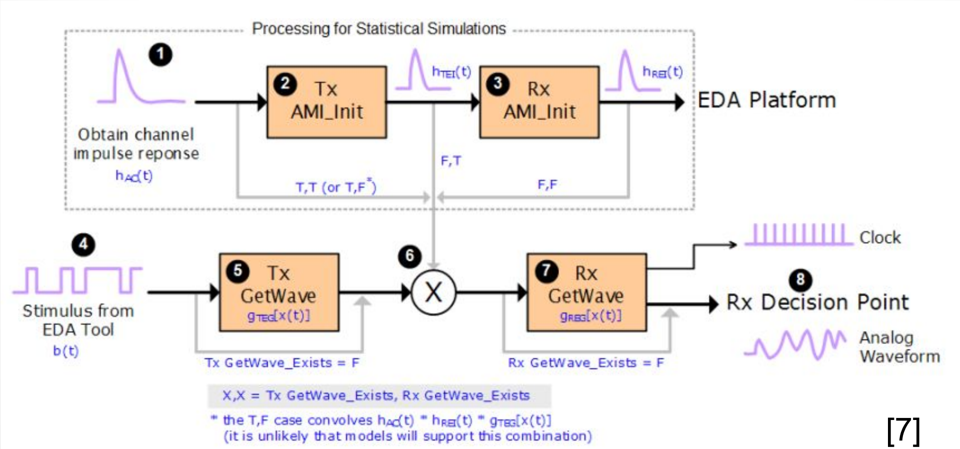
Call getWave
many times...

# **Analysis 2:** Co-optimization (internal)

- Convolution/FFT is “commutative”
- Use “delta response” in Tx ❷❺
  - Basically returns same wave_data
- Combine Tx & Rx in Rx’s proxy class ❸❼
  - Optimization separately or together
  - Combine “Model_Specific” parameters

$$(f*g)[n] \stackrel{\text{def}}{=} \sum_{m=-\infty}^{\infty} f[m]\,g[n-m]$$
$$= \sum_{m=-\infty}^{\infty} f[n-m]\,g[m]. \quad \text{(commutativity)}$$




[7]

# Analysis 2: code snippet

```
// Proxy's model has multiple stages
typedef struct ProxyMdl {
  void *txModl;
  void *rxModl;
};
// My own optimization routine
bool optimize(double *wavData, long  wavSize, char  **outParm, void *modlPtr);

/* The following is proxy class's getWave function, called by the simulator */
IBIS_AMI_API long AMI_GetWave(double *wavData, long wavSize, double *clkTime,
                              char  **outParm, void *modlPtr) {
  HMODULE txDll = LoadLibrary("C:/Temp/SPISimAMI_Tx.dll");
  HMODULE rxDll = LoadLibrary("C:/Temp/SPISimAMI_Rx.dll");
  amiGWav txGWav = (amiGWav) GetProcAddress(txDll, "AMI_GetWave");
  amiGWav rxGWav = (amiGWav) GetProcAddress(rxDll, "AMI_GetWave");
  if (txGWav && rxGWav) {
    // Proxy contains multiple stages
    void *txModl = ((ProxyMdl*)modlPtr)->txModl;
    void *rxModl = ((ProxyMdl*)modlPtr)->rxModl;

    // Optimize TX & RX until converged
    long txStat, rxStat;
    bool converged = false;
    while (!converged) {
      // call getWave of various stages
      txStat = (txGWav)(wavData, wavSize, clkTime, outParm, txModl);
      rxStat = (rxGWav)(wavData, wavSize, clkTime, outParm, rxModl);

      // call my own optimization routine, update param and check converged
      converged = optimize(wavData, wavSize, outParm, modlPtr);
      assert((txStat == 1) && (rxStat == 1));
    }
  }
  FreeLibrary(txDll);
  FreeLibrary(rxDll);
  return 1;
}
```

getWave calls of various stages

Self-optimization

# **Analysis 3:** Co-optimization (external)

- Use simulator's post-processing to get performance metrics
  - Simulator loads a proxy class
  - This proxy class serves as a "client" [8]
  - Query server and feeds data back to simulator
  - Simulator post-processes data and writes results to disk

- A standalone process (e.g. AMI test driver) loaded a proxy class [9]
  - This proxy class serves as a "server" (codes similar to Analysis 2)
  - Use socket, IPC or file to communicate with client
  - Persistent across multiple simulator invocations
  - Client response changes get updated continuously
    - Based on parameters passed by client (via socket), or
    - Based on post-processed results from simulator

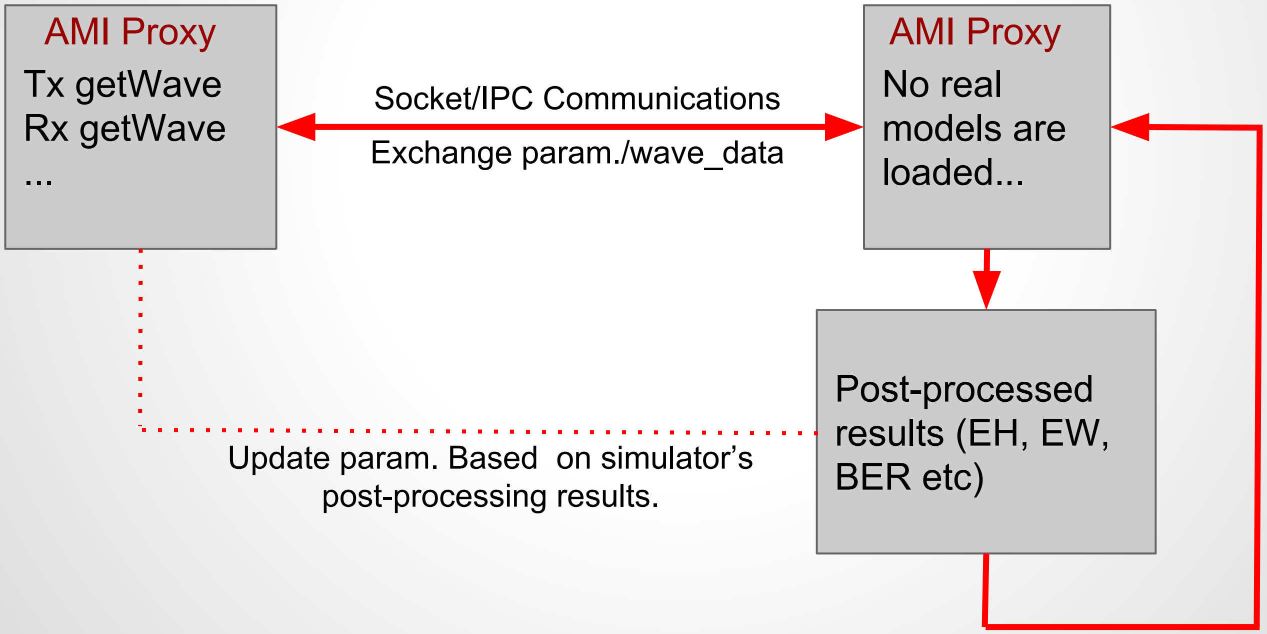
# **Analysis 3:** flow diagram

Stand-alone process

(Persistent, as a Server)

Simulator

(Invoked many times, as a Client)



# Analysis 3: client code snippet

```
/* The following is proxy class's getWave function, called by the simulator */
IBIS_AMI_API long AMI_GetWave(double *wavData, long  wavSize, double *clkTime,
                              char  **outParm, void *modlPtr) {
  // variables declarations, clean-up and error checks omitted below...

  // convert data to buffer to be sent to server
  byte *sendbuf = convDataToBuffer(wavData, wavSize, outParm);

  // Resolve the server address and port
  iResult = getaddrinfo(argv[1], DEFAULT_PORT, &hints, &result);

  // Attempt to connect to an address until one succeeds
  for (ptr = result; ptr != NULL ; ptr = ptr->ai_next) {
    // Create a SOCKET for connecting to server
    ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
    // Connect to server.
    iResult = connect(ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
    break;
  }
  freeaddrinfo(result);

  // Send an initial buffer
  iResult = send(ConnectSocket, sendbuf, (int)strlen(sendbuf), 0);

  // shutdown the connection since no more data will be sent
  iResult = shutdown(ConnectSocket, SD_SEND);

  // Receive data
  byte *recvbuf = recv(ConnectSocket, recvbuflen, 0);

  // cleanup
  closesocket(ConnectSocket);

  // convert receiving buffer data back to waveform for calling simulator
  convBufferToData(recvBuf, &wavData, &wavSize, &outParm);
  return 1;
}
```

AMI data to sending buffer

Socket communication

AMI data from receiving buffer

# Summary:

- A proxy class is useful for AMI development:
  - An AMI .dll(s)/.so(s) calling model's AMI .dll(s)/.so(s)
    - Can capture, post-processing and redirect waveform data
    - Can intercept calls and perform customized flow
    - Modularized to be independent with simulator and models
    - Can be applied to today's simulators and models

- Some useful testing/experiments:
    - Consistency and stress tests of AMI models
    - (Back-channel) co-optimization
      - Using internal or external process
    - Can integrate into either simulator or models easily

# References:


1. IBIS Spec. V6.1
2. Co-Optimization of SerDes Channels using AMI Modeling, June, 2014, IBIS Summit at San Francisco
3. The Backchannel Crossroads, June, 2014, IBIS Summit at San Francisco
4. BIRD 147.4 Back-channel Support draft 5
5. Proxy design pattern
6. Man in the middle
7. Simulating High-Speed Serial Channels with IBIS-AMI Models (KeySight)
8. Winsock client and server code sample
9. IBIS-AMI test drivers (e.g. from SPISim, SISoft and Cadence or HSpice's AMICheck)

# Q & A

SPISIM

EDA Expertise in Signal, Power Integrity & Simulation

SYNOPSYS
Integrated with
HSPICE®
Learn More >>

 SPISim is an InSync member.