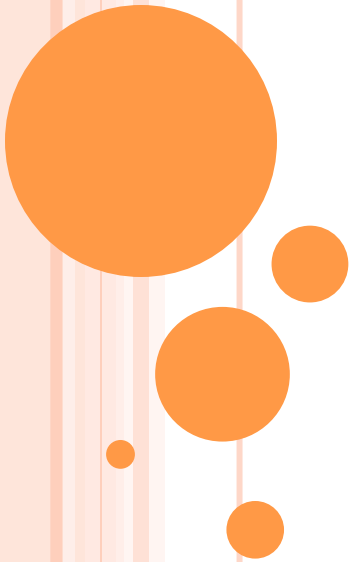


METROII AND PTOLEMYII INTEGRATION

Presented by: Shaoyi Cheng, Tatsuaki Iwata, Brad Miller, Avissa Tehrani

Introduction

- PtolemyII is a tool for design of component-based systems using heterogeneous modeling techniques.
  - PtolemyII studies modeling, simulation, and design of concurrent, real-time embedded systems.
  - Different MoCs such as Synchronous Dataflow (SDF) and Finite State Machines (FSM) can be constructed in PtolemyII.
  - Directors define how actors in the design fire and how tokens are used to communicate between them.

Introduction

- MetroII is a framework for platform-based design which allows functionality to be assigned to architecture.
  - A platform-based design methodology offers separation of concerns between architecture and functionality
  - By defining these two parts of a design through a set of clearly defined abstractions and systematically mapping functionality onto architecture, structured design space exploration is facilitated.

Motivation

- Developing functionality to interface PtolemyII with MetroII would allow more flexibility and efficiency in implementation of PtolemyII models.
- The objective of our project is to allow users to describe a system using the PtolemyII graphical user interface and to produce a MetroII description of the system which maintains the semantics of the original PtolemyII model.
- This means that designers are able to graphically build their functional models in PtolemyII and map them onto virtual architecture models in MetroII. This flow will allow developers to efficiently explore the design space in the early stages of development.

Motivation

- Currently, PtolemyII can generate C code for SDF, FSM and HDF models. The goal of this project is to add a module to the PtolemyII code generator to create MetroII descriptions of PtolemyII models. The MetroII descriptions can be used to implement the functionality described by the PtolemyII model on a particular architecture.

Scope of task

- In principle, there are many similarities between a MetroII and SystemC description of a system. As none of our group members are regular developers for MetroII, we preferred work on SystemC because it is more stable and better documented.
- We started with the Synchronous Dataflow domain, as the semantics of this domain seem easiest to maintain in SystemC. In order to simplify the design, we have temporarily restricted ourselves to designs which only require a size 1 buffer between the modules.

Analysis

Analysis

- Our SystemCCodeGenerater is based on CCodeGenerater of PtolemyII.
- CCodeGenerator uses “Helpers” and “Templates” for C. Reuse of this approach is possible if several key differences are addressed.
- Comparison of the C code generated by CCodeGenerator and handcrafted SystemC code done manually.
- Major differences
  - Variable declaration and initialization
  - Data transmission
  - Firing scheduling

Variable Declaration & Initialization

Generated C Code

- Based on procedural programming
- Variable declarations are generated in a group
- Initialization of parameters is also in a group

SystemC Code

- Based on Object-oriented Programming
- Variable declarations are in module declaration (SC_MODULE)
- Initialization process is called in constructor (SC_CTOR)

Code Comparison (Variable Declaration)

Generated C Code

```
/* Declaration of parameters */
int Actor1_param1;
int Actor2_param1;
….

/* Firing functions for each actor */
Actor1( ){
    set value;
}
Actor2( ){
    …
}

initialzie(){
    Actor1_param1 = 0;
    Actor2_param2 = 2;
     …
}
```

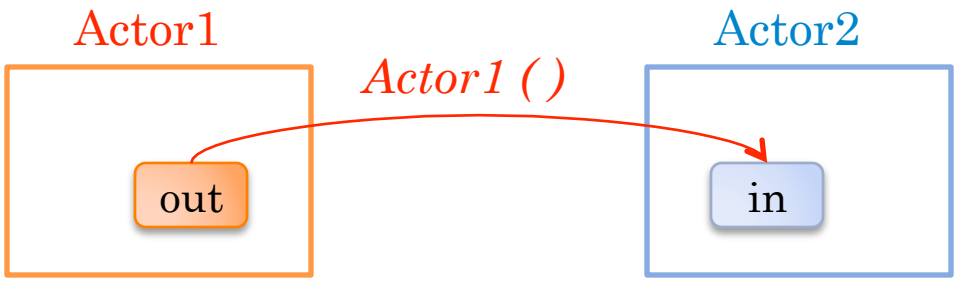
SystemC Code

```
SC_MODULE(Actor1){
    int param1;
    SC_CTOR(Actor1){
            SC_METHOD(fire( ));
            initialize( );
    }
}
Actor1::initialize( ){
    param1 = 0;
}

Actor1::fire( ){
    …
}
SC_MODULE(Actor2){
    int param2;
    SC_CTOR(Actor2){
            SC_METHOD(fire( ));
            initialize( ); }}
Actor2::initialize( ){  … }
Actor2::fire( ){ …. }
```

Data Transmission

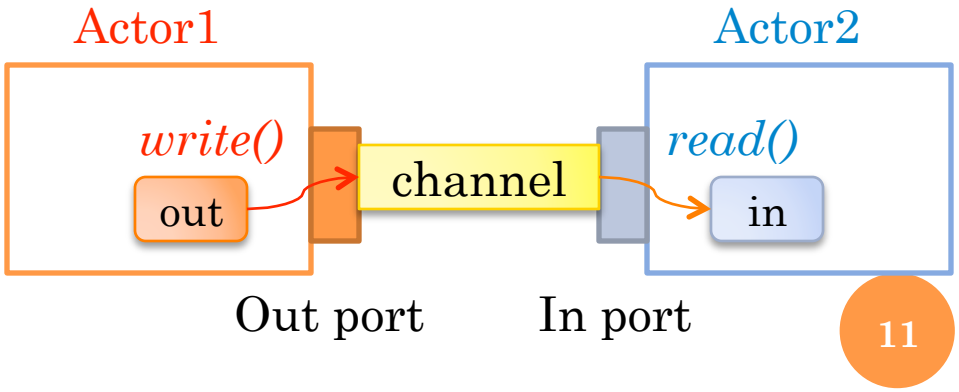
Generated C Code

- The preceding actor's fire function sets the following actor's input value directly.



SystemC Code

- Data is transferred through ports and channels.



Code Comparison (Data Transmission)

Generated C Code

```
/* input and output value */
int Actor1_out;
int Actor2_in;

/* firing function */
Actor1( ){
    Actor2_in = Actor1_out;
}
```

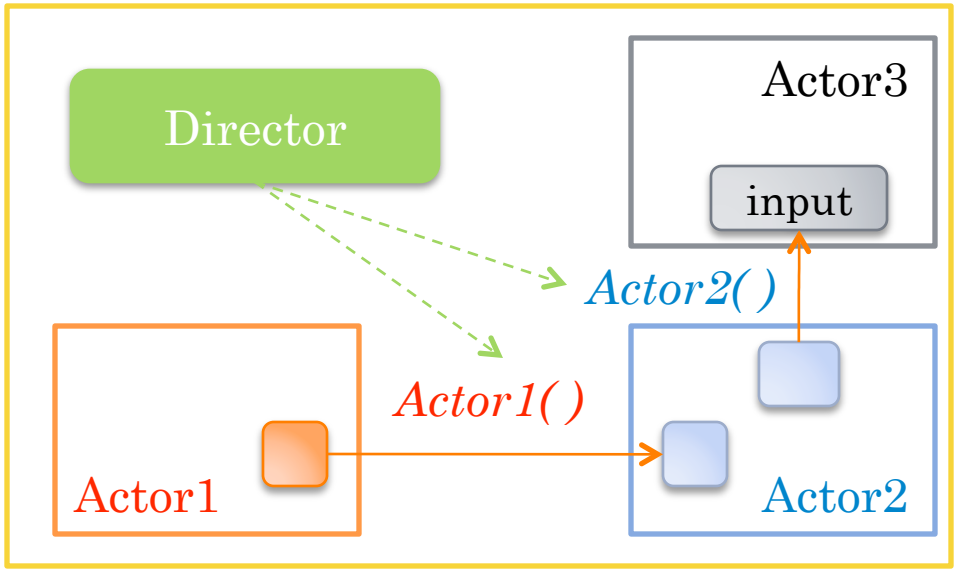
SystemC Code

```
SC_MODULE(Actor1){
    sc_out<int>  in_port;     int  a1_value;
}
Actor1::fire( ){
    out_port.write(a1_velue);
}
SC_MODULE(Actor2){
    sc_in<int>    out_port;    int a2_value;
}
Actor2:fire( ){
    a2_value= in_port.read();
}
void sc_main(){
    sc_buffer<int> wire;          // channel
    Actor1 Actor1_inst;           // instance
    Actor2 Actor2_inst;
    Actor1_inst.out_port(wire); // connection
    Actor2_ins.in_port(wire);
}
```

Firing Scheduling

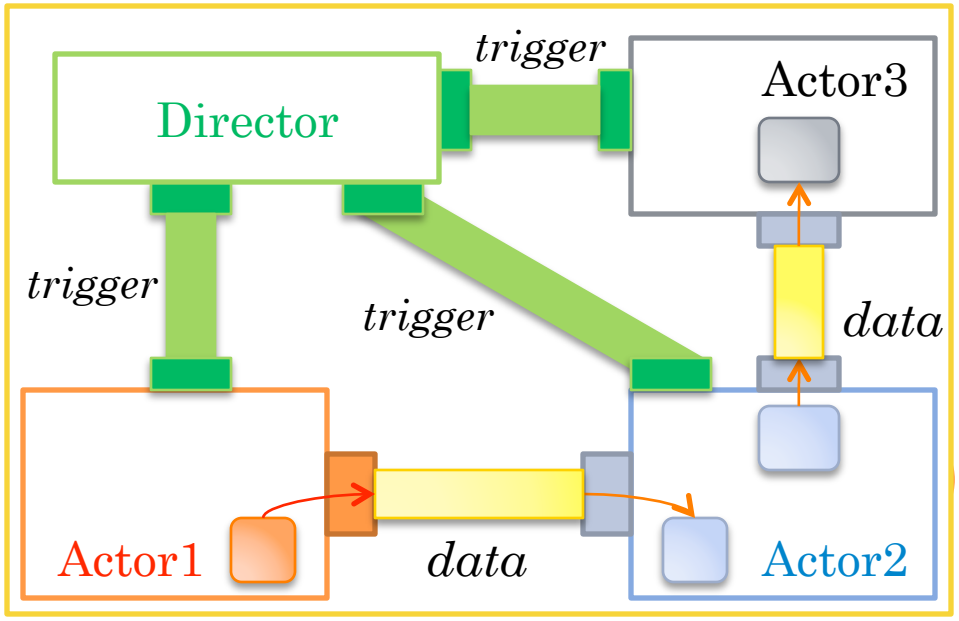
Generated C Code

- Fire functions of each actor are invoked in order assigned by Director.



SystemC Code

- Director module sends a trigger signal to a special "trigger" port of each actor.



Code Comparison (Firing Scheduling)

Generated C Code

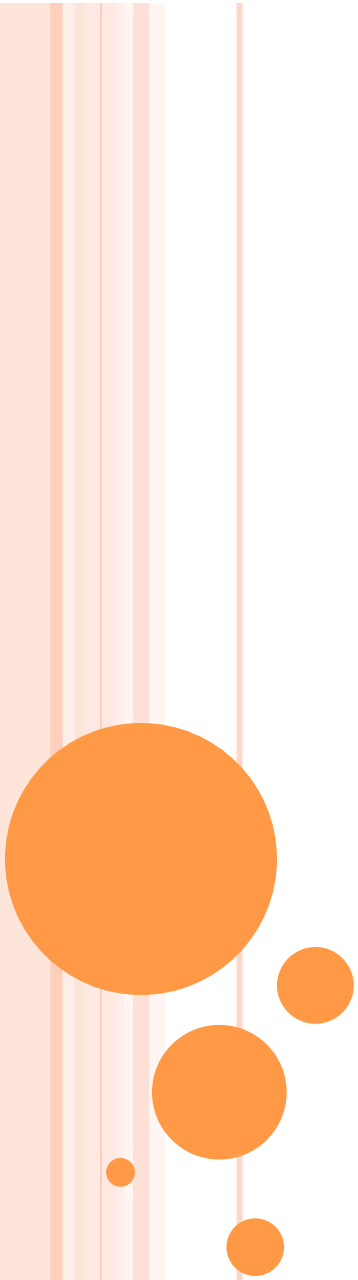
```
void main {
    int iteration=10; // get from model
    // firing in order scheduled by director
    for(int i=0; i<iterations; i++){
            Actor1( );
            Actor2( );
    }
}
```

SystemC Code

```
SC_MODULE(Director){
    sc_out<bool> fireTrigger[N];
}
Director::fire( ){
     for(int i=0; i<iterations; i++){
            fireTrigger.wirte(true);
    …
void sc_main(){
     /* trigger channel*/
    sc_buffer<bool> trigger[N];
    /* trigger connection */
    Director_inst.fireTrigger(trigger[0]);
    Actor1_inst.trigger(trigger[0]);
    Director_inst.fireTrigger(trigger[1]);
    Actor2_inst.trigger(trigger[1]);
    /* run simulation */
    sc_start();
}
```

Challenges in Development

- Variable Declaration & Initialization
  - Reallocate the declaration code per corresponding module.
- Data Transmission
  - Generate the code for I/O port of each actor.
  - Generate the declaration of the data channel and connection with ports.
- Firing Scheduling
  - Create director module declaration
  - Generate director's firing process in correct order.
  - Generate the declaration of the trigger channels and connection with trigger ports of every actor.



Design and Implementation

Overview

- Modularization
  - Tagging when code is generated for each actor
  - Regroup when code is ready for output
- Generation of connections between modules
  - Link different modules for data communication
- A separate director module (scheduler)
  - A director module which coordinates the dataflow
  - Fires each actor in the right order

Modularization

- Try to leverage the existing codegen infrastructure
- Problem: code was generated in sections (instead of modules) to match the procedural nature of C
  - Declaration
  - Initialization
  - Firing
- Each of the sections contains code from all actors
  - Need a way to break them down

Modularization

- Even though C is procedural, a Ptolemy model itself is modular
  - The original codegen has to traverse lists of actors for generation of operations for each of them
  - Each actor generates its own code for each section and the code segments are reassembled to complete each section
  - Modular structure can be restored
- Methodology employed
  - Tagging and regrouping

Modularization: Tagging

- For each section, a tag is added before code snippets from each actor are combined
- Information in tags
  - Instance name of actors
    - Necessary as multiple instantiation of the same actor can behave differently
  - Delimits the boundaries between pieces of code from different actors

Modularization: Regrouping

- Code segments are re-parsed and re-organized before output
  - Search for tags to extract the operation of each actor for each section
  - Group all sections generated by an actor together
- Module declarations and function name declaration are added to wrap around the generated code

Modularization: Example

Original:

```
int actor1_output;
int actor2_output;
void initialize(){
    actor1_output =  0;
    actor2_output =  0;
}
void actor1()
{
    actor1_output +=1;
}
void actor2()
{
    actor_output += 2;
}
…
```

Tagged:

```
/***actor1***/
int actor1_output;
/***end actor1 ***/
/***actor2***/
int actor2_output;
/***end actor2***/
void initialize()
{   /***actor1***/
    actor1_output =  0;
    /***end actor1***/
    /***actor2***/
    actor2_output =  0;
    /***end actor2***/
}
void actor1()
{   /***actor1***/
    actor1_output +=1;
    /***end actor1***/
}
…
```

Regrouped:

```
SC_MODULE(actor1)
{
    int actor1_output;
    void initialize();
    void fire();
    SC_CTOR(actor1)
    {…}
}
void actor1::initialize()
{
    actor1_output =  0;
}
void actor1::fire()
{
    actor1_output +=1;
}
…
```

Generation of connections

- In C code generation all variables are global
  - Sender changes the value of receiver port directly
  - Fire code of the sender
- SystemC is modular
  - No direct change of value of internal members from another actor
  - Connection is needed to pass data value

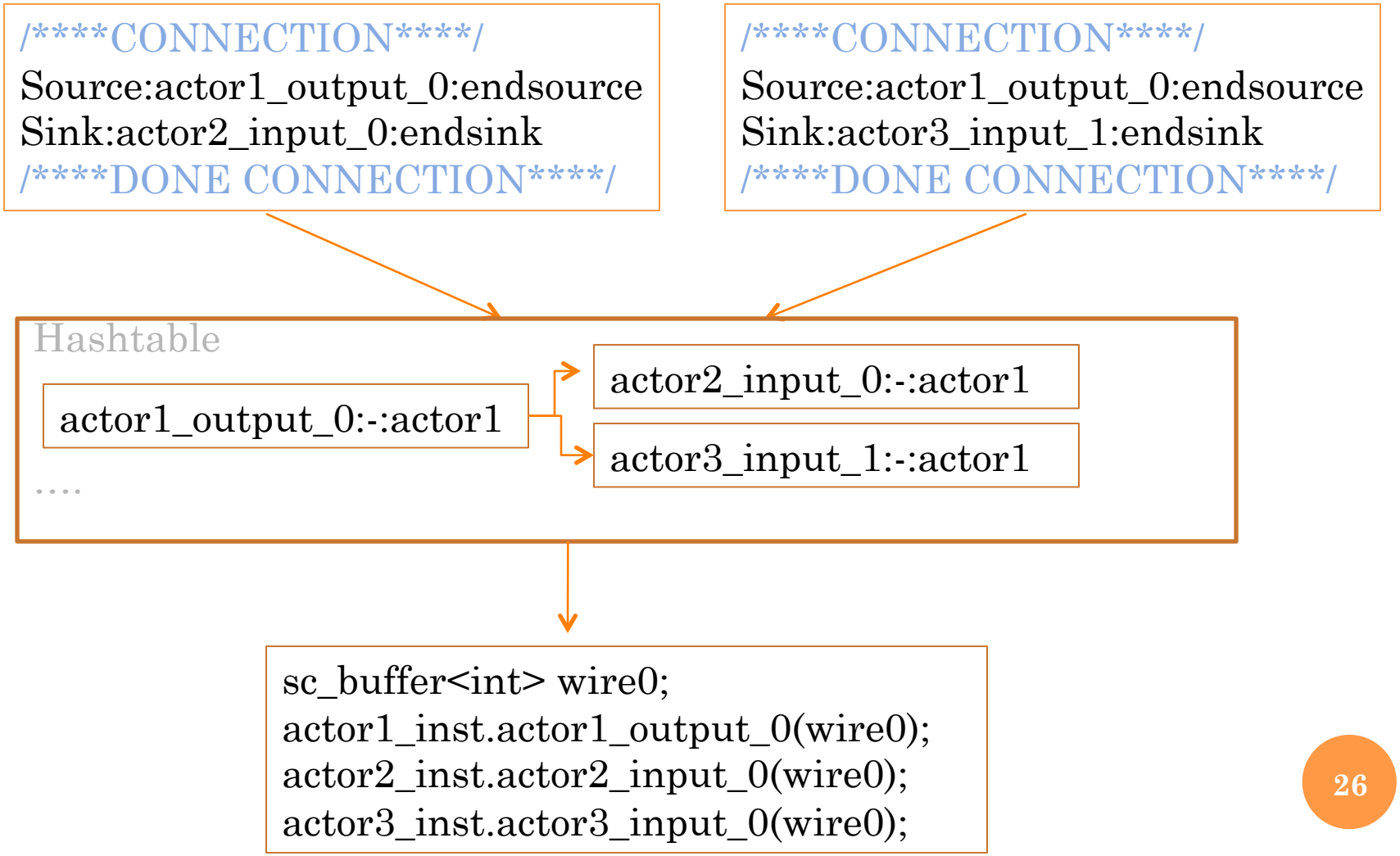
GENERATION OF CONNECTIONS

- Tagging is used again when fire code is being generated
  - Find source and sink for each connection
    - Parse fire code to look for assignment statement
  - Tag a connection and annotate variable as source/sink
  - Every connection can have one source and multiple sinks
- Before output, the tagged code is reparsed to generate a hashtable

GENERATION OF CONNECTIONS

- The hashtable is keyed using concatenation of the source port name and the container actor, and contains the value which points to the sink
  - The source port is made an output port of the sender module
  - The sink port is made an input port of the receiver module
- Wires are instantiated for each connection when modules are instantiated

GENERATION OF CONNECTIONS: EXAMPLE



A Separate Director (Scheduler)

- Instead of a simple for loop in the original C codegen, a separate scheduler is created
  - Driven by clock signal and triggers each actor in the right order
  - Manages max iteration count and increments the iteration index each clock tick
- Generated by the corresponding director class in the Ptolemy model
  - Director class has access to the iteration count

A Separate Director (Scheduler)

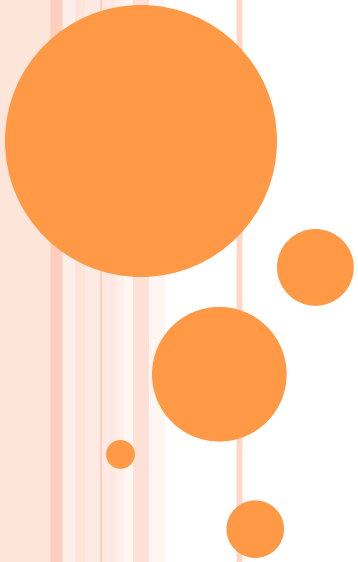
- Acquiring the firing sequence from the generated fire code
  - The original C codegen creates fire code in the right order
  - Actor names embedded in the tags are used to extract the sequence for firing actors
- Pulls the trigger of each actor to fire
  - Communication and control for each actor separated
  - Trigger is a special port for control

A Separate Director: Example

```
sc_buffer<bool>TRIGGER[ACTOR_NUM];
_Ramp_SDF_Director_Ramp_SDF_Director_inst("_Ramp_SDF_Director");

_Ramp_SDF_Director_inst.trigger(CLOCK);

_Ramp_SDF_Director_inst.fireTrigger[0](TRIGGER[0]);
_Ramp_SDF_Director_inst.fireTrigger[1](TRIGGER[1]);
_Ramp_SDF_Director_inst.fireTrigger[2](TRIGGER[2]);

Ramp_Ramp_inst.trigger(TRIGGER[0]);
Ramp_CompositeActor_SequencePlotter_inst.trigger(TRIGGER[1]);
Ramp_SequencePlotter_inst.trigger(TRIGGER[2]);
```

Director driven by clock

Director firing actors in order

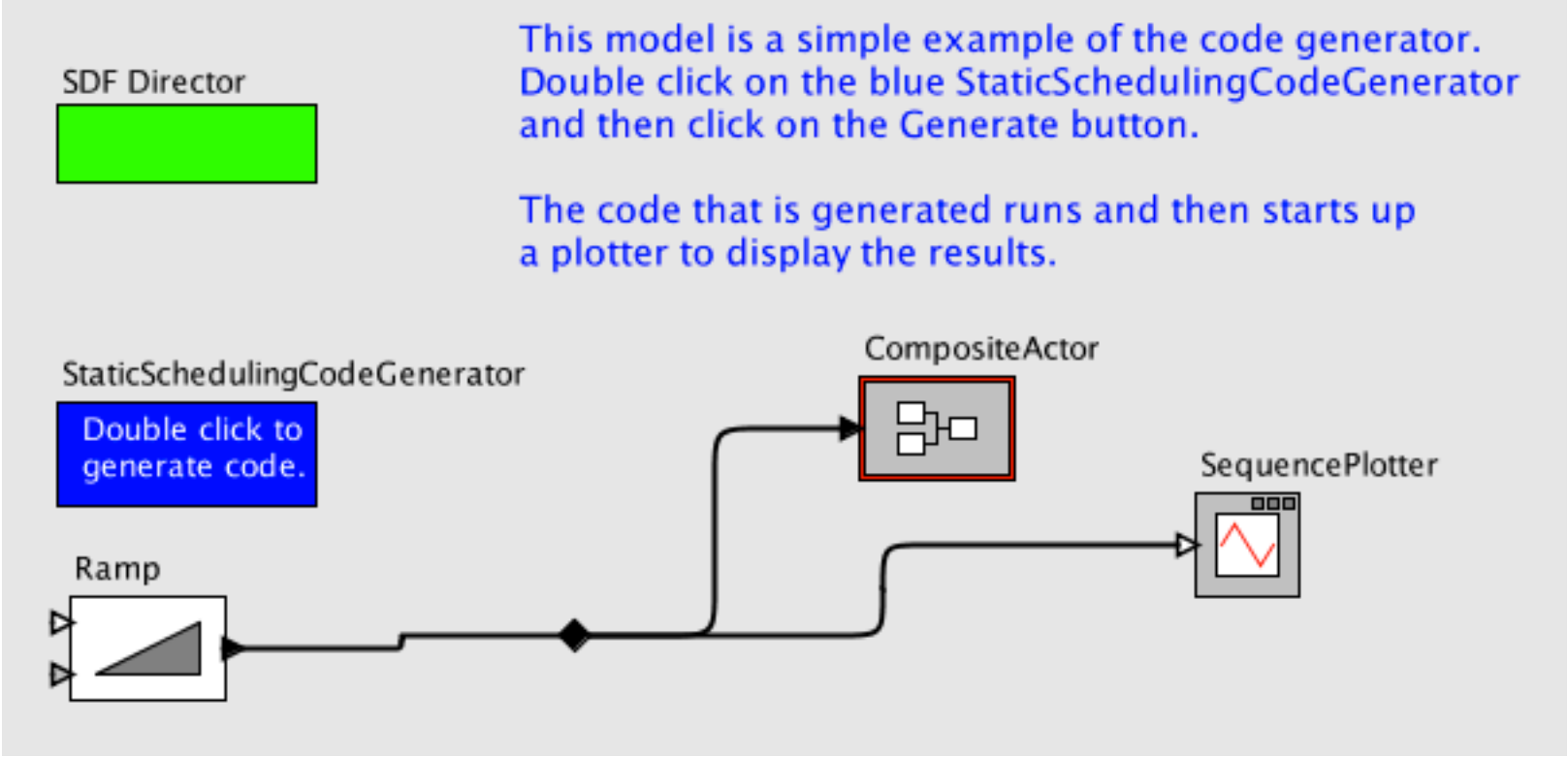


Code Generation Examples

Available Actors

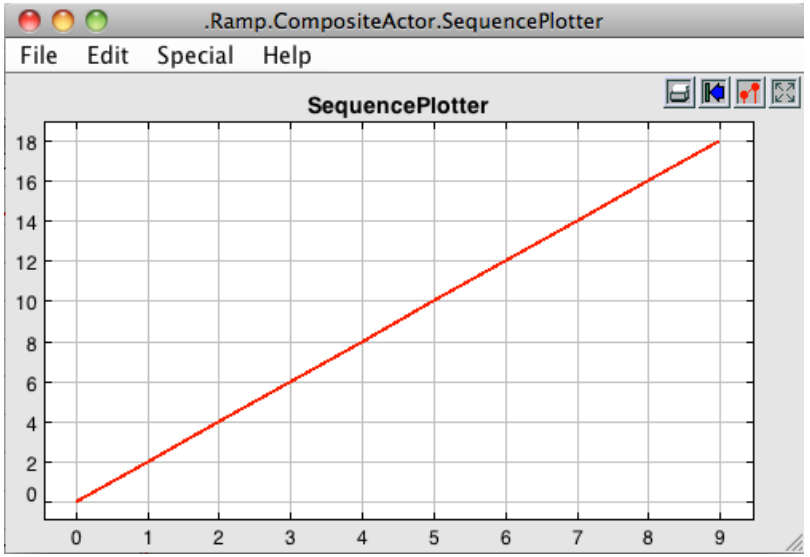
- Ramp
- Sequence Plotter
- Composite Actor
- Add/Subtract
- Absolute Value

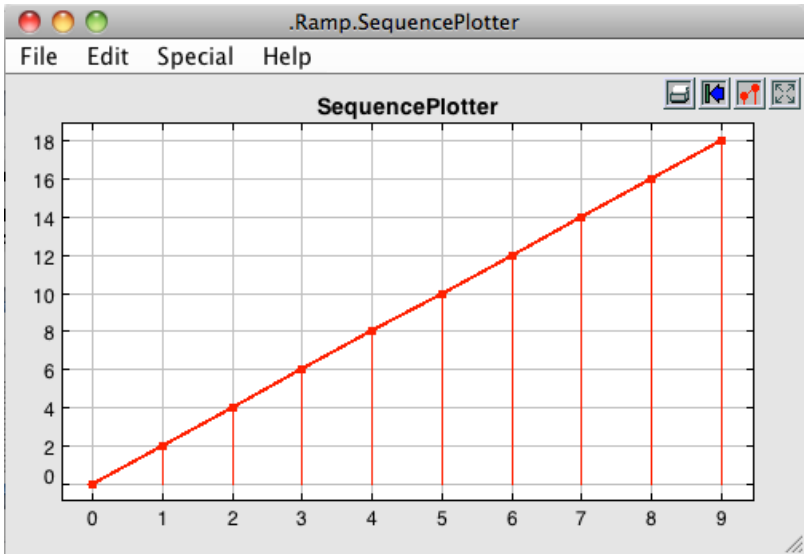
Modified Ramp Demo



Modified Ramp Demo: Output

```
bmiller1@ubuntu:~/courses/ee249/demos$ ./Ramp.bin

        SystemC 2.2.0 --- Dec  3 2009 17:51:01
    Copyright (c) 1996-2006 by all Contributors
                ALL RIGHTS RESERVED
Ramp_CompositeActor_SequencePlotter__output 0: 0
Ramp_SequencePlotter__output 0: 0
Ramp_CompositeActor_SequencePlotter__output 1: 2
Ramp_SequencePlotter__output 1: 2
Ramp_CompositeActor_SequencePlotter__output 2: 4
Ramp_SequencePlotter__output 2: 4
Ramp_CompositeActor_SequencePlotter__output 3: 6
Ramp_SequencePlotter__output 3: 6
Ramp_CompositeActor_SequencePlotter__output 4: 8
Ramp_SequencePlotter__output 4: 8
Ramp_CompositeActor_SequencePlotter__output 5: 10
Ramp_SequencePlotter__output 5: 10
Ramp_CompositeActor_SequencePlotter__output 6: 12
Ramp_SequencePlotter__output 6: 12
Ramp_CompositeActor_SequencePlotter__output 7: 14
Ramp_SequencePlotter__output 7: 14
Ramp_CompositeActor_SequencePlotter__output 8: 16
Ramp_SequencePlotter__output 8: 16
Ramp_CompositeActor_SequencePlotter__output 9: 18
Ramp_SequencePlotter__output 9: 18
SystemC: simulation stopped by user.
```





Modified Ramp Demo: Actor Code

```
/* Actor: Ramp_Ramp */
SC_MODULE(Ramp_Ramp) {
   sc_out<int> Ramp_Ramp_output_0_port;
   /* trigger input */
   sc_in<bool> trigger;
   /* Ramp's referenced parameter declarations. */
   int Ramp_Ramp_step_;
   /* Ramp's type convert variable declarations. */
   int Ramp_Ramp_output_0;
   …
};

void Ramp_Ramp::initialize() {
   /* Ramp's parameter initialization */
   Ramp_Ramp_step_ = 2;
   /* initialize Ramp */
   Ramp_Ramp__state = 0;
}

void Ramp_Ramp::fire() {
   /* Fire Ramp */
   Ramp_Ramp_output_0 = Ramp_Ramp__state;
   Ramp_Ramp__state += Ramp_Ramp_step_;
   Ramp_Ramp_output_0_port.write
      (Ramp_Ramp_output_0);
   …
}
```

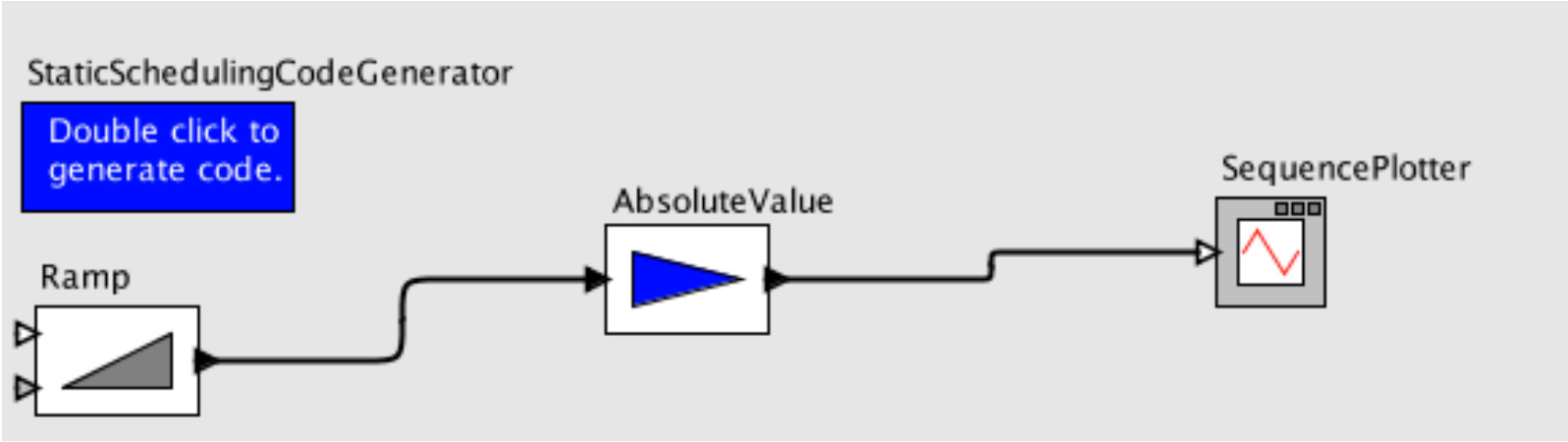
```
/* Actor: Ramp_SequencePlotter */
SC_MODULE(Ramp_SequencePlotter) {
  sc_in<int>Ramp_SequencePlotter_input_0__port;
   /* trigger input */
   sc_in<bool>trigger;
   /* SequencePlotter's input variable declarations. */
   double Ramp_SequencePlotter_input[1];
   /* preinitSequencePlotter */
   double Ramp_SequencePlotter__xValue;
   …
};

void Ramp_SequencePlotter::initialize() {
   /* initSequencePlotter */
   Ramp_SequencePlotter__xValue = 0.0;
}

void Ramp_SequencePlotter::fire() {
   Ramp_SequencePlotter_input[0] = ((double)
      Ramp_SequencePlotter_input_0__port.read());
   /* Fire SequencePlotter */
   cout << "Ramp_SequencePlotter__output" << "
      "<<Ramp_SequencePlotter__xValue << ": "
      <<Ramp_SequencePlotter_input[0]<<"\n";
   Ramp_SequencePlotter__xValue++;
}
```

Modified Ramp Demo: Director & Main

```
/* main part */
int sc_main(int argc, char *argv[]) {
    sc_clock    CLOCK("clock");
    sc_buffer<bool>TRIGGER[ACTOR_NUM];
    _Ramp_SDF_Director_Ramp_SDF_Director_inst
        ("_Ramp_SDF_Director");
    _Ramp_SDF_Director_inst.trigger(CLOCK);
    _Ramp_SDF_Director_inst.fireTrigger[0](TRIGGER[0]);
    _Ramp_SDF_Director_inst.fireTrigger[1](TRIGGER[1]);
    _Ramp_SDF_Director_inst.fireTrigger[2](TRIGGER[2]);
    Ramp_RampRamp_Ramp_inst("Ramp_Ramp");
    Ramp_Ramp_inst.trigger(TRIGGER[0]);
    Ramp_CompositeActor_SequencePlotterRamp
        _CompositeActor_SequencePlotter_inst
        ("Ramp_SequencePlotter");
    Ramp_CompositeActor_SequencePlotter_inst.trigger
        (TRIGGER[1]);
    Ramp_SequencePlotterRamp_SequencePlotter_inst
        ("Ramp_CompositeActor_SequencePlotter");
    Ramp_SequencePlotter_inst.trigger(TRIGGER[2]);
    sc_buffer<int> wire0;
    Ramp_Ramp_inst.Ramp_Ramp_output_0_port(wire0);
    Ramp_SequencePlotter_inst.Ramp_SequencePlotter
        _input_0__port(wire0);
    Ramp_CompositeActor_SequencePlotter_inst.Ramp
        _CompositeActor_SequencePlotter_input
        _0__port(wire0);
    sc_start();
    exit(0);
}
```

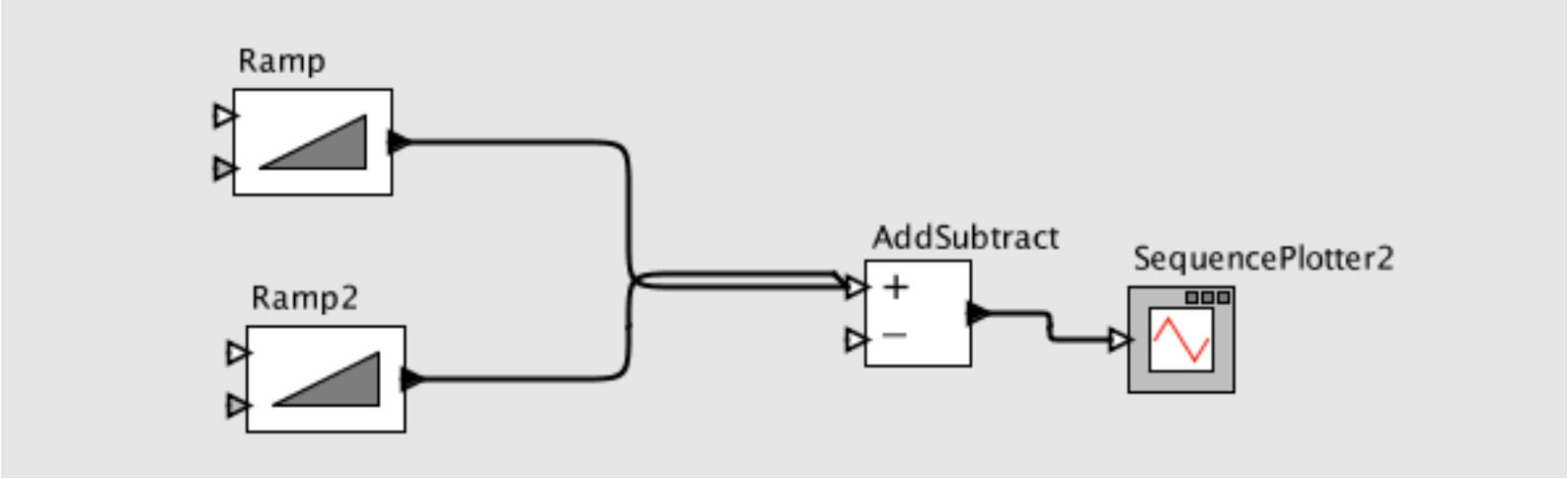
```
/* Director: _Ramp_SDF_Director */
SC_MODULE(_Ramp_SDF_Director) {
    /* clock trigger input */
    sc_in<bool>trigger;
    /* fire trigger output */
    sc_out<bool>fireTrigger[ACTOR_NUM];
    int_count;
    int iterations;
    …
};

void _Ramp_SDF_Director::initialize() {
    iterations = 10;
    _count = 0;
}

void _Ramp_SDF_Director::fire() {
    while(true) {
        for(int i=0; i<ACTOR_NUM; i++) {
            fireTrigger[i] = true;
            wait(SC_ZERO_TIME);
        }
        _count++;
        if(_count >= iterations) {
            sc_stop();
        }
        wait();
    }
}
```

Absolute Value Demo




```
bmiller1@ubuntu:~/courses/ee249/demos$ ./abs_with_neg.bin

        SystemC 2.2.0 --- Dec  3 2009 17:51:01
    Copyright (c) 1996-2006 by all Contributors
                ALL RIGHTS RESERVED
abs_SequencePlotter__output 0: 5
abs_SequencePlotter__output 1: 3
abs_SequencePlotter__output 2: 1
abs_SequencePlotter__output 3: 1
abs_SequencePlotter__output 4: 3
abs_SequencePlotter__output 5: 5
abs_SequencePlotter__output 6: 7
abs_SequencePlotter__output 7: 9
abs_SequencePlotter__output 8: 11
abs_SequencePlotter__output 9: 13
SystemC: simulation stopped by user.
```

Adder Demo




```
bmiller1@ubuntu:~/courses/ee249/demos$ ./
    Ramp2.bin

   SystemC 2.2.0 --- Dec  3 2009 17:51:01
Copyright (c) 1996-2006 by all Contributors
        ALL RIGHTS RESERVED
Ramp2_SequencePlotter2__output 0: 3
Ramp2_SequencePlotter2__output 1: 6
Ramp2_SequencePlotter2__output 2: 9
Ramp2_SequencePlotter2__output 3: 12
Ramp2_SequencePlotter2__output 4: 15
Ramp2_SequencePlotter2__output 5: 18
Ramp2_SequencePlotter2__output 6: 21
Ramp2_SequencePlotter2__output 7: 24
Ramp2_SequencePlotter2__output 8: 27
Ramp2_SequencePlotter2__output 9: 30
```

Design Challenges

- Multiple inputs
  - Incompatible with native data-driven control
  - Addressed in our design by trigger ports
- Multiple tokens on single buffer
  - Would prohibit simply connecting actor data ports with wires
  - Buffers would need to be statically sized
  - Synchronization would be needed between actors and buffers so that no data is lost

Conclusion

- This project is an attempt to extend PtolemyII code generator to generate a MetroII description
- We restricted our scope to SDF model and SystemC code generator as a first version of our work.
- Our initial step of code modification was to modify the C code generator to output only the firing code for each actor.
- The SystemC code generated by this project is similar to MetroII code