Integrated Finite State Machine and Dataflow Modeling
Outline

- Introduction
- Integrated Modeling
- Introduction to SysteMoC Programming
- Analysis of Integrated Models
- Summary
- Hands-on SysteMoC
Dataflow Graphs

- Dataflow graphs are widely accepted for modeling DSP algorithms
  - Multimedia
  - Signal processing

- Well known optimization strategies for static dataflow (SDF) graphs

- Modeling complex multimedia applications using SDF graphs only is challenging or even impossible
Dynamic Dataflow Graphs

- Dynamic dataflow (DDF) graphs are well suited for modeling complex multimedia and signal processing applications.

- Ad hoc representation of control flow in DDF graphs has led to problems in analyses.

- In summary, neither SDF graphs nor DDF graphs alone are an appropriate foundation to base a design flow upon.
Integrated FSM and Dataflow Modeling

- Actors often model control-dominant behavior
- Finite State Machine (FSM) are an appropriate modeling approach for control-dominant applications
- Concurrency models based on hierarchical FSM are often ambiguous
- Consequently: Integrated FSM and Dataflow Modeling has attracted a lot of attention in the recent years
Finite State Machines

Definition (Finite State Machine): An *Finite State Machine (FSM)* is a five-tuple \( M = (S, \Sigma, \Lambda, \tau, s_0) \) with
- \( S \) being a finite set of states,
- \( \Sigma \) being a set of symbols denoting possible inputs,
- \( \Lambda \) being a set of symbols denoting possible outputs,
- \( \tau : S \times \Sigma \rightarrow S \times \Lambda \) being a state transition function, and
- \( s_0 \in S \) being the initial state.

- In one *reaction*, an FSM maps a current state \( s \) and an input symbol \( \sigma \in \Sigma \) to a next state \( s' \) and an output symbol \( \lambda \in \Lambda \). Hence, \( \tau (s, \sigma) = (s', \lambda) \).
- Straight forward extension to multiple signals possible
  - \( n \) input signals: \( \Sigma = \Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_n \)
  - \( m \) output signals: \( \Lambda = \Lambda_1 \times \Lambda_2 \times \ldots \times \Lambda_m \)
State Transition Diagram

- Graphical representation of FSMs
- Vertices represent states \( s \in S \).
- Edges represent state transitions \( \tau \).
  - Transitions are labeled with guard/action pairs, where guard \( \in \Sigma \) and action \( \in \Lambda \).
  - A state transition is selected from all enabled transitions.
  - A transition is enabled if it is represented by an outgoing edge from the vertex representing the current state and the guard matches the current input symbol.

<table>
<thead>
<tr>
<th>current state</th>
<th>( s_0 )</th>
<th>( s_0 )</th>
<th>( s_1 )</th>
<th>( s_1 )</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>input symbol</td>
<td>( \sigma_2 )</td>
<td>( \sigma_1 )</td>
<td>( \sigma_1 )</td>
<td>( \sigma_2 )</td>
<td>...</td>
</tr>
<tr>
<td>next state</td>
<td>( s_0 )</td>
<td>( s_1 )</td>
<td>( s_1 )</td>
<td>( s_0 )</td>
<td>...</td>
</tr>
<tr>
<td>output symbol</td>
<td>( \varepsilon )</td>
<td>( \lambda_1 )</td>
<td>( \varepsilon )</td>
<td>( \lambda_2 )</td>
<td>...</td>
</tr>
</tbody>
</table>
Outline

- Introduction
- Integrated Modeling
  - Specify dataflow actors using FSMs
  - Control dataflow graphs by FSMs
- Introduction to SysteMoC Programming
- Analysis of Integrated Models
- Summary
- Hands-on SysteMoC
Several options exist how to integrate FSM and dataflow modeling:

- Specify dataflow actors using FSMs,
- Control dataflow graphs by FSMs, or
- Refine states by dataflow graphs

In this lecture, the first two options are covered

We start with a simple approach
Simple Actors

**Definition (Actor):** An actor is a three-tuple \( a = (I, O, M) \) containing actor ports, partitioned into actor input ports \( I \) and actor output ports \( O \), and an actor FSM \( M \).

- Ports will be used to connect actors to communication channels
- Actors only communicate via channels (Actor-oriented modeling)
- Actor FSM \( M \) specifies the behavior of the actor
Example Simple Actor

- $I = \{i_1, i_2\}$
- $O = \{o_1\}$
- Actor FSM... later
Definition (Network Graph): A network graph is a directed graph \( G = (A, Q, \pi) \) containing a set of actors \( A \), a set of FIFO channels \( Q \subseteq A.O \times A.I \) with infinite buffer size, and a channel parameter function \( \pi : Q \rightarrow V^* \), which associates with each channel \( q \in Q \) a possibly non-empty sequence \( v \in V^* \) of initial tokens.

- Furthermore, all actor ports \( A.I \cup A.O \) are restricted to be connected to exactly one channel \( q \in Q \).
- FIFO channels supporting
  - Blocking, destructive reads
  - Non-blocking, non-destructive writes
Example Simple Network Graph

- $A = \{a_1, a_2, a_3\}$
- $Q = \{(a_1.o_1, a_3.i_1), (a_2.o_1, a_3.i_2), (a_3.o_1, a_1.i_1), (a_3.o_2, a_2.i_1)\}$
- $\pi(a_3.o_1, a_1.i_1) = \pi(a_3.o_2, a_2.i_1) = "1"
- Channel parameter functions are omitted for channels without initial tokens
Definition (Actor FSM): The *actor FSM* $M$ of an actor $a \in A$ is an FSM $(S, \Sigma, \Lambda, \tau, s_0)$ with input alphabet $\Sigma = 2^{a.I}$ and output alphabet $\Lambda = 2^{a.O}$.

- $a.I$ denotes the set of input ports of actor $a$
- $a.O$ denotes the set of output ports of actor $a$
- $2^X$ denotes the power set of $X$
  - $X = \{x_1, x_2\} \Rightarrow 2^X = \emptyset, \{x_1\}, \{x_2\}, \{x_1, x_2\}$
- The input and output symbols specify the consumption and production of tokens, respectively
Example (Cyclo Static Dataflow)
FOR each actor DO
  FOR each state transition originating in the current state DO
    Evaluate guard
    Non-deterministically select a transition with satisfied guard
    Consume tokens according to the guard
    Produce tokens according to the action
    Update current state
  ENDFOR
ENDFOR
ENDFOR
Example (Cyclo Static Dataflow)
So far introduced simple actor model cannot express data-dependent behavior

In the following, each token carries a value $v \in V$.

Channels are queues supporting

- Blocking, destructive reads (consumption) and non-blocking, non-destructive writes (production) in FIFO order
- Non-blocking, non-destructive reads (peak) and non-blocking, destructive writes (poke) in random order
Guards (1/3)

- Guards (input symbols $\sigma \in \Sigma$) are now given as formulas in first order logics consisting of a conjunction of
  - Consumption rates, where $c(i, n)$ specifies to consume $n \in \mathbb{N}$ tokens from the channel connected to the input port $i \in I$,
  - Production rates, where $p(i, m)$ specifies to produce $m \in \mathbb{N}$ tokens onto the channel connected to the output port $o \in O$, and
  - Comparisons $i[k] = v$ of token values with constant values $v \in V$. The $[]$-operator returns the value of the $k$-th token in the channel connected to the input port $i \in I$. 
Guards (2/3)

- Only token values from tokens specified in the consumption rates are allowed to be compared.
- Indices start with 0

Example
- Correct: $c(i_1, 2) \land c(i_2, 1) \land (i_2[0] = 2)$
- Incorrect: $c(i_1, 2) \land c(i_2, 1) \land (i_2[1] = 2)$
Guards (3/3)

A guard evaluates to true if

- Each specified consumption rate $c(i, n)$ is satisfied, i.e., at least $n$ tokens are available from the channel connected to the input port $i$,
- Each specified production rate $p(o, m)$ is satisfied, i.e., $m$ additional token can be stored in the channel connected to output port $o$ (always true in presence of infinite queues), and
- Each specified comparison is true

\[
c(i_1, 2) \land p(o_1, 2) \land (i_1[0] = 2) \land (i_1[1] = 0)
\]
Actions (1/2)

- Actions (output symbols $\delta \in \Delta$) are now given as sequences of assignments $\langle (o_1[0] := 1), (o_1[1] := 2), \ldots \rangle$
- The $[]$-operator returns a reference of the $k$-th token to be produced to the channel connected to the output port $o \in O$. 

The diagram shows a transition from state $s_0$ to state $i_1$ on input $a_2$, with guard conditions $c(i_1, 2) \land p(o_1, 2) \land (i_1[0] = 2) \land (i_1[1] = 0)$ and transitions $\langle (o_1[0] := 2), (o_1[1] := 4) \rangle$. The output $o_1$ transitions to state $3$. 

© University of Erlangen-Nuremberg
Christian Haubelt
Actions (2/2)

- A transition is enabled if its guard evaluates to true
- If an enabled transition is executed
  - Additional token are generated according to the production rates,
  - The action is performed (values are assigned to the new tokens according to the action),
  - Tokens are consumed and produced according to the consumption and production rates, and
  - The state of the actor FSM is updated
Example Data-Dependent Behavior

\( a_1 \)

\( a_2 \)

\( a_3 \)

1. \( c(i_1, 1) \land p(o_1, 1) \land o_1[0] := 1 \)
2. \( c(i_1, 1) \land p(o_1, 1) \land o_1[0] := 2 \)
3. \( c(i_1, 1) \land p(o_1, 1) \land p(o_2, 1) \land (i_1[0] = 1) \land (i_2[0] = 2) \)
Outline

- Introduction
- Integrated Modeling
  - Specify dataflow actors using FSMs
  - Control dataflow graphs by FSMs
- Introduction to SysteMoC Programming
- Analysis of Integrated Models
- Summary
- Hands-on SysteMoC
Idea

- Refine a single actor by a dataflow graph
- Control the firing of the actors of the dataflow graph by an FSM

- A refined actor is called a cluster
- The FSM is called the cluster FSM

- A cluster without cluster FSM is a network graph
- A cluster without refining dataflow graph is an actor
Definition (Cluster): A cluster is a six-tuple $a_c = (I, O, A, Q, \pi, M)$ containing cluster ports, partitioned into cluster input ports $I$ and cluster output ports $O$, a set of actors $A$, a set of FIFO channels $Q \subseteq A.O \times A.I$ with infinite buffer size, and a channel parameter function $\pi: Q \rightarrow V^*$, which associates with each channel $q \in Q$ a possibly non-empty sequence $v \in V^*$ of initial tokens, and a cluster FSM $M$.

- The cluster FSM is an actor FSM where the action is a static schedule of actors in the refining dataflow graph.
Example SDF Cluster (1/2)
Example SDF Cluster (2/2)
CSDF Schedule (1/2)

\[
c(i_1, 2) \land p(o_1, 2) / \langle a_2 \rangle
\]

\[
c(i_2, 2) \land p(o_1, 2) / \langle a_1, a_1, a_3, a_3 \rangle
\]
CSDF Schedule (2/2)

c(i_1, 2) \land p(o_1, 2) / \langle a_2 \rangle

c(i_2, 2) \land p(o_1, 2) / \langle a_1, a_3, a_3 \rangle
Outline

- Introduction
- Integrated Modeling
- Introduction to SysteMoC Programming
  - Formal Model
  - Programming
- Analysis of Integrated Models
- Summary
- Hands-on SysteMoC
SysteMoC Actors

- So far presented modeling approach not applicable for real-world applications
- Additional abstractions are mandatory

**Definition (Actor):** An *actor* is a four-tuple $a = (I, O, F, M)$ containing actor ports partitioned into actor input ports $I$ and actor output ports $O$, the actor functionality $F$ and an actor FSM $M$.

- The actor functionality can be seen as an actor refinement not necessarily restricted to dataflow models
Example SysteMoC Actors

- \( I = \{i_1, i_2\} \)
- \( O = \{o_1\} \)
- Actor functionality ... next
- Actor FSM ... later
SysteMoC Actor Functionality

Definition (Actor Functionality): The *actor functionality* is three-tuple \( F = (F_a, F_g, Z) \), where \( F_a \) is a finite set of action functions, \( F_g \) is a finite set of guard functions, and \( Z \) is a finite set of states.

- Action functions and guard functions must terminate eventually.
- Action functions are only allowed to change the current state \( z \in Z \) of the actor functionality and the token values to be produced on the output channels.
- Guard functions are Boolean-valued functions that must not change the state of the model at all. This includes:
  - The current state \( z \in Z \) of the actor functionality \( a.F \),
  - The current state \( s \in S \) of the actor FSM \( a.M \) , and
  - The number and the values of tokens in the channels connected to the actor.
Example SysteMoC Actor Functionality

- $F_a = \{f_1, f_2\}$
- $F_g = \{g_1\}$
- $Z$

- Action functions and guard functions are triggered by the actor FSM
SysteMoC Actor FSM

Definition (Actor FSM): The actor FSM is a three-tuple $M = (S, T, s_0)$ consisting of a finite set of states $S$, a finite set of transitions $T$, and an initial state $s_0 \in S$.

- A transitions $t = (s, \text{cons}, \text{prod}, f_g, f_a, s') \in T$ from the current state $s \in S$ to a next state $s' \in S$ defines
  - its input consumption rate $\text{cons} : I \rightarrow \mathbb{N}_0$
  - its output production rate $\text{prod} : O \rightarrow \mathbb{N}_0$.

- The guard function $f_g \in F_g$ is a Boolean-valued expression over the values of the tokens required on the input ports and the current state $z \in Z$ of the actor functionality.

- The action function $f_a \in F_a$ determines the values of the tokens, which are to be produced on the outputs.
Example SysteMoC Actor FSM

\[ A = \{ s_0, s_1 \} \]

\[ T = \{ (s_0, ((i_1, 1))), ((o_1, 1)), g_1, f_1, s_1),
  (s_1, ((i_2, 1))), ((o_1, 1)), \emptyset, f_2, s_0) \} \]
SysteMoC Actor Behavior

- Actors are composed of a Finite State Machine (FSM), functions, and variables
- The FSM controls the function invocation
- Functions are executed atomically
- Data consumption and production is performed at the end of an action execution

<table>
<thead>
<tr>
<th>Actor state</th>
<th>active</th>
<th>passive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Actor FSM</td>
<td>Variables</td>
</tr>
<tr>
<td></td>
<td>call</td>
<td>read</td>
</tr>
</tbody>
</table>

Functions:
- Action Functions
- Guard Functions

Diagram:
- Actor FSM
- Variables
- Functions
- Action Functions
- Guard Functions
Simulation Semantics

- A transition is active if
  - All referred input queues have enough tokens,
  - All referred output queues have enough free space,
  - And the guard function evaluates to true

- An active transition may be fired
  - The associated action function is atomically executed
  - Consumption and production of data tokens take place
  - Dataflow between actors may activate or deactivate actions

- If several transitions of one actor are active
  - One transition is selected non-deterministically
Example Square Root Calculation

\[ c(i_1, 1) \land p(o_1, 1) / f_{\text{in}} \]

\[ c(i_1, 1) \land p(o_1, 1) \land g_{\text{exact}} / f_{\text{loop}} \]

\[ c(i_2, 1) \land p(o_1, 1) / f_{\text{in}} \]

\[ c(i_1, 1) \land p(o_2, 1) \land g_{\text{exact}} / f_{\text{out}} \]
Outline

- Introduction
- Integrated Modeling
- Introduction to SysteMoC Programming
  - Formal Model
  - Programming
- Analysis of Integrated Models
- Summary
- Hands-on SysteMoC
SysteMoC – SystemC library for Dataflow Modeling

- **SysteMoC**
  - Elementary channels separating functionality and communication
  - C++ syntax for specifying actor FSMs
  - Scheduler for (dynamic) dataflow MoC domain

- **Elementary Channels**
  - Signal, timer, mutex, semaphore, FIFO, etc.

- **Core Language**
  - Modules
  - Ports
  - Processes
  - Events
  - Interfaces
  - Channels

- **Data-Types**
  - 4-valued logic types (01zx)
  - 4-valued logic vectors
  - Bits and bit-vectors
  - Arbitrary-precision integers
  - Fixed-point numbers
  - C++ user-defined types

- **Event-driven Simulation Kernel**

- **C++ Language Standard**
Hello World – Objectives

- You will see a state-of-the-art “Hello World” application ...
- ... modeled in SysteMoC.
Hello World (hello.cpp)

```cpp
#include <iostream>
#include <systemoc/smoc_moc.hpp>

class HelloActor: public smoc_actor {

public:
    // actor constructor
    HelloActor(sc_module_name name)
        : smoc_actor(name, start) {
        // FSM definition:
        // transition from start to end calling action src
        start = CALL(HelloActor::src) >> end;
    }

private:
    smoc_firing_state start , end; // FSM states
    void src() { // action
        std::cout << "Actor " << this->name() << " says :
                   
                   " << "Hello World" << std::endl;
    }
};
```
SysteMoC Actors

- **Include SysteMoC library header**
  ```cpp
  #include <systemoc/smoc_moc.hpp>
  ```

- **An actor is a C++ class derived from base class**
  ```cpp
  class HelloActor: public smoc_actor {
  ```

- **An action function is a method of the actors class**
- **Actions shall be private (prohibit execution by others)**
  ```cpp
  private:
  void src() { // action
    std::cout << "Actor "
    << this->name () << " says :\n"
    << "Hello World" << std::endl;
  }
  ```
SysteMoC Actors

- An actor has a finite set of states
  
  ```
  smoc_firing_state state_a , state_b;
  ```

- Like objects of a C++ classes, we create instances of an actor

- Constructors are responsible for creating a certain actor instance

- Provide actor name and start state to base class

  ```
  smoc_actor
  HelloActor(sc_module_name name)
  : smoc_actor(name , state_a) {
  ```
A finite state machine (FSM) defines transitions between states

E.g. a transition from state_a to state_b

If this transition is taken, the action HelloActor::src is executed

```c++
HelloActor(sc_module_name name) :
  : smoc_actor(name, state_a) {
    state_a = CALL(HelloActor::src) >> state_b;
}
```
class HelloNetworkGraph : public smoc_graph {
public:
    // network graph constructor (create actor HelloWorld)
    HelloNetworkGraph(sc_module_name name)
    : smoc_graph(name), helloActor("HalloActor")
    {
    }
private:
    // actors
    HelloActor helloActor;
};

int sc_main(int argc , char ** argv) {
    // create network graph
    HelloNetworkGraph top("top");
    smoc_scheduler_top sched(top);
    sc_start(); // start simulation (SystemC)
    return 0;
}
A network graph is derived from base class Smoc_graph

```cpp
class HelloNetworkGraph : public Smoc_graph {

    private:
        HelloActor helloActor;

    // A name for the graph has to be passed to base class Smoc_graph
    // We need to call HelloActor's constructor
    HelloNetworkGraph(sc_module_name name)
        : Smoc_graph(name),
          helloActor("HelloActor")
    {
    }
```

Our network graph has an instance of the HelloActor
Simulation

- **Function** `sc_main` **is the entry point for simulation (cf. SystemC)**
- The network graph is instantiated
- Parameters are passed to the graph constructor (e.g.“top”)
- **Call** `sc_start` **to start simulation (cf. SystemC)**

```c
int sc_main(int argc, char ** argv) {
    HelloNetworkGraph top("top");
    smoc_scheduler_top sched(top);
    sc_start();
    return 0;
}
```
Compilation and Execution

- Compile the source code (hello.cpp) using e.g., gcc
- Run simulation (OSCI SystemC: execute binary for simulation)

```bash
./hello
```

Simulation output

```
SystemC 2.2.0 --- Dec 15 2008 11:10:07
Copyright (c) 1996-2006 by all Contributors
ALL RIGHTS RESERVED
Actor top.HalloActor says:
Hello World
SystemC: simulation stopped by user.
```
class Sink: public smoc_actor {
public:
    // ports:
    smoc_port_in <char> in;
    Sink(sc_module_name name) // actor constructor
        : smoc_actor(name, start) {
        // FSM definition:
        start = in(1) >>
            CALL(Sink::sink) >> start;
    }
private:
    smoc_firing_state start; // FSM states
    void sink() {
        std::cout << this->name () << " recv: \"
        << in[0] << "\" << std::endl;
    }
};
Ports and Channels – Input Port

- Create an input port
- Ports have a data type (e.g. `char`)
  
  ```cpp
  smoc_port_in <char> in;
  ```

- Declare to read one token in FSM transition
  ```cpp
  start = in(1) >>
         CALL(Sink::sink) >> start;
  ```

- Write data in action
  ```cpp
  void sink () {
    std::cout << this->name () << " recv: '\""
               << in[0] << '\"' << std::endl;
  }
  ```
class Source: public smoc_actor {
public:
    // ports:
    smoc_port_out <char> out;
    Source(sc_module_name name) : smoc_actor(name, start) {
        start = out(1) >>
            CALL(Source::src) >> start;
    }
private:
    smoc_firing_state start; // FSM states
    void src() {
        std::cout << this->name () << " send: \'X\'" << std::endl;
        out[0] = 'X';
    }
};
Ports and Channels – Output Port

- Create an output with data type `char`

```cpp
smoc_port_out<char> out;
```

- Declare to write one token in FSM transition

```cpp
start = out(1) >> CALL(Source::src) >> start;
```

- Access data within actions

```cpp
void src() {
    std::cout << this->name () << " send: \'X\'" << std::endl;
    out[0] = 'X';
}
```
class NetworkGraph : public smoc_graph {
public:
    NetworkGraph(sc_module_name name) // constructor : smoc_graph(name),
        source("Source"), // create actors
        sink("Sink") {
        connectNodePorts (source.out , sink.in);
    // connect actors
    }
private:
    Source source; // actors
    Sink sink;
};

int sc_main(int argc , char ** argv) {
    smoc_top_moc <NetworkGraph> top("top"); // create network graph
    sc_start(); // start simulation (SystemC)
    return 0;
}
Ports and Channels – FIFO Queues

- Connect a pair of ports (input, output) using a FIFO queue
- Connected ports have to use the same data type
- Queues have default size “1” (one data token)
  ```
  connectNodePorts(source.out, sink.in);
  ```

- Set queue size explicitly
  ```
  connectNodePorts<23>(source.out, sink.in);
  ```

- Initial data tokens later
Ports and Channels

➢ Simulation output

```
SystemC 2.2.0 --- Dec 15 2008 11:10:07
Copyright (c) 1996-2006 by all Contributors
ALL RIGHTS RESERVED

top.Source send: 'X'
top.Sink recv: 'X'
top.Source send: 'X'
top.Sink recv: 'X'
top.Source send: 'X'
top.Sink recv: 'X'
...
```

➢ Simulation runs infinitely
The source actor produces two data tokens per invocation

We declare to produce $n$ data tokens in a transition using `out(n)`

```plaintext
start = out(2) >>
    CALL(Source::src) >> start;
```

Use the `[ ]`-operator to write data values

```plaintext
void src () {
    out[0] = message[count++];
    out[1] = message[count++];
}
```

Similar to arrays addressing range is $0, \ldots, n-1$
Communication Rates – Avoid Deadlocks

- Simulation output (using Sink actor from previous example)

  SystemC 2.2.0 --- Dec 15 2008 11:10:07
  Copyright (c) 1996-2006 by all Contributors
  ALL RIGHTS RESERVED
  SystemC: simulation stopped by user.

- Nothing happens! Why?
- Writing two tokens requires free space for (at least) two tokens
- We need to increase the queue size (implicit size was “1”)
- Minimum size of 2 is mandatory (but actors would run in lockstep)
- Using larger sized queues may decouple execution of actors

  `connectNodePorts<4>(source.out, sink.in);`
Initial Tokens – Syntax

- Explicit creation of a FIFO initializer
- Give queue size as constructor parameter
  ```
  smoc_fifo<int> initFifo(1);
  ```
- Push initial value to initializer
  ```
  initFifo << 42;
  ```
- Pass initializer when constructing the queue
  ```
  connectNodePorts(source.out, sink.in, initFifo);
  ```
- Initializer may be reused for creating identical initialized queues
Guards and Actions – Source Actor

```cpp
static const std::string MESSAGE = "0123456789";

class Source: public smoc_actor {
public:
    smoc_port_out <char> out;
    Source(sc_module_name name) : smoc_actor(name, start), 
        count(0), size(MESSAGE.size()), message(MESSAGE) {
        start = GUARD(Source::hasToken) >> 
            out(1) >> 
            CALL(Source::src) >> start;
    }
private:
    smoc_firing_state start;
    unsigned int count, size; // variables (functional state)
    const std::string message; // 
    bool hasToken() const{ return count < size; } // guard 
    void src() { out[0] = message[count ++]; } // action
};
```
Guards and Actions – Guards

- A guard is a `const` member function returning a Boolean value
  
  ```cpp
  bool hasToken() const{
    return count < size;
  }
  ```

- Guards enable/disable transitions (true/false)
- Guards must (can) not change variable values or token in channels
- Refer to guards via `GUARD( . . )` macro

  ```cpp
  start = GUARD(Source::hasToken) >>
           out(1) >>
           CALL(Source::src) >> start;
  ```

- Use guards for control flow (see below)
Guards and Actions – Variables

- Variables ...
- ... are private class member of an actor
- ... can be used to store data
- ... represent a functional state of an actor (in contrast to FSM state)

```cpp
unsigned int count, size;
const std::string message;
```
Guards and Actions – Actions

- Actions ...
- ... are used to read/write data on input/output ports
- ... modify variables

```c
void src() {
    out[0] = message[count ++];
}
```

- Guards access variables read-only (mandatory `const` modifier)
- Actions are allowed to modify variables
Guards and Actions

- Simulation output (using Sink actor from previous example)

```
top.Sink recv: "0"
top.Sink recv: "1"
top.Sink recv: "2"
top.Sink recv: "3"
top.Sink recv: "4"
top.Sink recv: "5"
top.Sink recv: "6"
top.Sink recv: "7"
top.Sink recv: "8"
top.Sink recv: "9"
SystemC: simulation stopped by user.
```

- Source actor sends a finite number of characters only
- Simulation terminates when no actor can be activated
Outline

- Introduction
- Integrated Modeling
- Introduction to SysteMoC Programming
- Analysis of Integrated Models
- Summary
- Hands-on SysteMoC
Hierarchy of Streaming Models

- HSDF: Homogeneous SDF
- SDF: Synchronous Dataflow
- CSDF: Cyclo-Static Dataflow
- BDF: Boolean Dataflow
- KPN: Kahn Process Network
- DDF: Dynamic Dataflow
- RPN: Reactive Process Network

BDF and larger: Turing complete

[Staten@MoCC2008]
Classification

- Classify actors by
  - Minimizing the communication FSM
  - Pattern matching

- Note that actor classification in its general form is undecidable

- In the following only static dataflow models are considered that can be recognized:
  - SDF (Synchronous Data Flow)
  - CSDF (Cyclo-Static Data Flow)
Simple Example

- Most basic representation (left figure) of an actor, which can be classified to be an SDF actor (right picture)
- A single state with one outgoing state transition
Actor state

- Problem: Actor FSM describes the possible communication behavior of the actor
- However, only a subset of transitions may be enabled during the execution of the actor (data dependencies)
- To accommodate both, FSM state and functionality state of an actor, we define an actor state as the pair $x = (z, s)$
  - $Z$ is the (possibly infinite) set of functionality states
  - $S$ is the finite set of FSM states
- When an enabled transition is executed, the next actor state $x'$ is defined as follows:
  - The next FSM state $x'.s$ is destination state of the transition
  - The next functionality state $x'.z$ is the new functionality state after the execution of the associated action
Dynamic state transition diagram

- $S = \{ s_0 \}$
- $Z = \{ (0,0), (1,0), (0,1), (1,1) \}$
- The actor state space $Z \times S$ induces a diagram, which refines the possible communication behavior described by the FSM.
Dynamic State Transition Diagram

For each actor state $x = (z, s)$ and each enabled transition $\tau$:

- Add edge $e$ to dynamic state transition diagram connecting $x$ with $x'$. 
- Annotate $e$ with the number of tokens consumed and produced by $\tau$ as vectors.
Classification Algorithm

- Classification requirements:
  - Liveliness, i.e., each actor state must have at least one enabled outgoing transition (SDF / CSDF actors can always be activated if enough tokens available)
  - Active input / output behavior, i.e., no infinite sequence of edges, which do not consume or produce any tokens

- If requirements are not met, deadlocks could be introduced by treating the actor as an SDF or CSDF actor

- Assume we have a hypothesis $\gamma$ („classification candidate“)
  - $\gamma.\nu$: Number of phases
  - $\gamma.cons$: Maps phase to vector of consumed tokens
  - $\gamma.prod$: Maps phase to vector of produced tokens

- Accept / Reject $\gamma$ by annotating $x.\rho$, $x.cons$, $x.prod$ to a visited state
Example Classification (1/3)

\[ \gamma_1 \cdot \nu := 1, \quad \gamma_1 \cdot \text{cons}(1) := (1), \quad \gamma_1 \cdot \text{prod}(1) := (0) \]
- failed
Example Classification (2/3)

\[ \gamma_1 \cdot \nu := 1, \quad \gamma_1 \cdot \text{cons}(1) := (1), \quad \gamma_1 \cdot \text{prod}(1) := (0) \ (\text{discarded}) \]

\[ \gamma_2 \cdot \nu := 1, \quad \gamma_2 \cdot \text{cons}(1) := (1), \quad \gamma_2 \cdot \text{prod}(1) := (1) \]
Example Classification (3/3)

- $\gamma_1.\nu := 1, \gamma_1.\text{cons}(1) := (1), \gamma_1.\text{prod}(1) := (0)$ (discarded)
- $\gamma_2.\nu := 1, \gamma_2.\text{cons}(1) := (1), \gamma_2.\text{prod}(1) := (1)$ (accepted)
- $\gamma_3.\nu := 2, \gamma_3.\text{cons}(2) := (1), \gamma_3.\text{prod}(2) := (0)$ (not needed)
Classification Algorithm Limitations

- Problem: Functionality state of actor may be
  - Not known in advance
  - Infinite
  - Non-deterministic

- Dynamic state transition diagram cannot be constructed/analyzed in such a case

- In order to be able to classify those actors anyhow, one can compute an over-approximation

- This leads to a state transition diagram based on the FSM

- Classification algorithm can also be applied to this diagram
Outline

- Introduction
- Integrated Modeling
- Introduction to SysteMoC Programming
- Analysis of Integrated Models
- Summary
- Hands-on SysteMoC
Summary

- Limited expressiveness of static dataflow graphs is not sufficient to model state-of-the-art multimedia or signal processing applications.
- Limited analyzability of dynamic dataflow graphs restricts their usability in automatic design flows.
- Integrated FSM and dataflow graph model helps to overcome some of these drawbacks.
- SysteMoC is an actor-oriented system programming language based on an integrated FSM and dataflow graph model.
- During classification of SysteMoC actors, “hidden” functionality state must be considered (or abstracted).
Integrated FSM and Dataflow Modeling

Contact information:
Christian Haubelt
Hardware/Software Co-Design
University of Erlangen-Nuremberg, Germany
http://www12.cs.fau.de/people/haubelt
haubelt@cs.fau.de

With contributions from
Joachim Falk, Jens Gladigau, Joachim Keinert,
Martin Streubühr, Christian Zebelein, and Jürgen Teich