Heterogeneous Function Composition in Embedded Software Synthesis to Eliminate Direct Relations Between Components

Pieter J. Mosterman
Senior Research Scientist Design Automation Department Adjunct Professor School of Computer Science

The problem!
- Demo ...

What happened?!

Agenda
- Simulink preliminaries
  - Executing a Simulink model
  - Dealing with hierarchy
  - Heterogeneous composition
  - An implementation in Simulink
  - Conclusions

Basic Simulink syntax
- An example model ...

sin = \sum_{z=1}^{\text{out}} f(z)
**Basic Simulink syntax**

- Network of blocks with directed lines connected between ports

![Diagram of Simulink blocks](image)

**Example block implementation**

- A unit delay

```cpp
class UnitDelayBlock : public Block {
  public:
    ErrorStatus BlockDrawIcon() {
      // Draw 'z^-1' on the icon
      return ErrorStatus::Ok;
    }

    BlockParameterData BlockGetParameterData() {
      // Return initial_condition as block data
      BlockParameterData data;
      data.initial_condition = 0;
      return data;
    }

    ErrorStatus BlockOutput() {
      // Implement y(t) = x(t)
      return ErrorStatus::Ok;
    }

    ErrorStatus BlockUpdate() {
      // Implement x(t) = u(t)
      return ErrorStatus::Ok;
    }

  private:
    double initial_condition;
};
```

**The execution hierarchy**

- The lines reflect input/output relations
- The (nonvirtual) blocks are dynamic systems

![Diagram of execution hierarchy](image)

**The basic structure of a discrete time block**

- Data dependencies between
  - Input signals, output signals, current state, new state, update function and output function

![Diagram of discrete time block](image)
Data dependencies to create a sorted list for efficient execution

- Static dependency analysis

Algebraic dependencies

- Strongly connected components

Algebraic dependencies

- Strongly connected components

Algebraic dependencies

- What if we replace the delay block by a gain?

\[ y(t) = u(t) + K y(t) \]

\[ y(t) = u(t) / (1 - K) \]
Agenda

- Simulink preliminaries
- Executing a Simulink model
- Dealing with hierarchy
- Heterogeneous composition
- An implementation in Simulink
- Conclusions

Let’s assume single-tasking execution

- Allows multi-rate systems
- All blocks run in a single task
  - Single execution time line
  - Base rate at greatest common denominator
  - Blocks execute when they have a sample hit
  - No data integrity issues

A simulation algorithm for networks of dynamic systems

First generate the sorted list

- Determine data dependencies
  - Based on a direct feedthrough (df) flag
Generate execution lists for all of the block methods

Repeatedly evaluate the execution lists

Agenda

Hierarchy

Virtual blocks to organize graphical hierarchy
  - Referential transparency; no semantic bearing

Nonvirtual blocks to organize
  - Execution hierarchy
  - Data scope hierarchy
A graphical hierarchy

- Group blocks in a virtual subsystem
  - Subsystem C does not appear in the sorted list

Let’s create a component …

- Make the virtual subsystem nonvirtual
  - Becomes a dynamic system in its own right
  - Does that affect sorting?

Let’s start with a virtual subsystem …

- Dependencies derived from df flag

Make subsystem an atomic component

- Direct feedthrough moves to component level!
Make subsystem an atomic component

- Now we have a dependency cycle!

How did the direct feedthrough flag end up there?! And how can we fix it?!

Gain followed by delay

Gain followed by delay

Agenda

- Simulink preliminaries
- Executing a Simulink model
- Dealing with hierarchy
- Heterogeneous composition
- An implementation in Simulink
- Conclusions
Gain followed by delay

How do we create a component?

Homogeneous composition

Put the component in context
We have a dependency cycle!

Heterogeneous composition!

Is there another way ... ?

Putting it in context again

No cycles!
Agenda

- Simulink preliminaries
- Executing a Simulink model
- Dealing with hierarchy
- Heterogeneous composition
- An implementation in Simulink
- Conclusions

How do we fit this into the model compilation machinery?

- Create sorted list after clearing df flag

- Mark where output for update should be used

What does the generated code look like?

```
/* Model output function */
void gaindelay_output(void)
{
  real_T rtb_E;

  /* Outputs for atomic SubSystem: '<Root>/C' */
  gaindelay_C();
  /* end of Outputs for SubSystem: '<Root>/C' */
  
  /* Gain: '<Root>/E' */
  rtb_E = gaindelay_P.E_Gain * gaindelay_B.Delay;
  
  /* Sum: '<Root>/B' incorporates: */
  /* Constant: '<Root>/A' */
  gaindelay_B.B = gaindelay_P.A_Value + rtb_E;
} /* to do */
```
What does the generated code look like?

```c
/* Model output function */
void gaindelay_output(void)
{
    real_T rtb_E;

    /* Outputs for atomic SubSystem: '<Root>/C' */
    gaindelay_C();

    /* Gain: '<Root>/E' */
    rtb_E = gaindelay_P.E_Gain * gaindelay_B.Delay;

    /* Sum: '<Root>/B' incorporates:
        * Constant: '<Root>/A' */
    gaindelay_B.B = gaindelay_P.A_Value + rtb_E;
}

/* Outputs for atomic system: '<Root>/C' */
void gaindelay_C(void)
{
    /* UnitDelay: '<S1>/Delay' */
    gaindelay_B.Delay = gaindelay_DWork.Delay_DSTATE;
}
```

The general analysis

- For an input block
  - Depth-first search for a df path to output
  - If a port without df is reached, mark visited nodes for potential move of output method
- If output found
  - Clear all blocks visited from the initial input port

The synthesis part of the algorithm

- Move marked df blocks into an atomic subsystem
The synthesis part of the algorithm

- Move marked df blocks into an atomic subsystem

Nested cyclic dependencies

Sorted list:

Output execution list:

- Applies to derivative and zerocrossing as well
- But, the df path has to be breakable

In general, four basic data dependency classes
Agenda

- Simulink preliminaries
- Executing a Simulink model
- Dealing with hierarchy
- Heterogeneous composition
- An implementation in Simulink
- Conclusions

Conclusions

- Simulink as a network of dynamic systems
  - Method set to generate behavior (output, update, …)
- Lists for the execution phases
  - df to sort according to data dependencies
  - Block methods and sample times to create lists
- Atomic subsystems for componentization
  - Heterogeneous composition to avoid dependencies
  - Heterogeneous execution lists (flat and hierarchical)
  - Scoped the class of systems where this applies

References


Acknowledgments

John E. Ciolfi
MathWorks
Ben Denckla
Independent Thinker

Many thanks for their continuing collaboration!