ECE 587 – Hardware/Software Co-Design
Lecture 06 Process-Based Models II

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

February 2, 2015
Project I

- Due: 10:00am 03/06 Chicago time
- Start working on it now. Don’t wait until the last week.
- Read “Guide to System Setup and Work Flow” on course website first.
Reading Assignment

- This lecture: 3.1.2, 3.2
- Next lecture: Project I, SystemC User Guide (see Blackboard)
Outline

Data Flow Graphs

System Design Languages

C++ Review
Data Flow Graphs (DFG)

- Could be treated as a special SDF
  - Each actor consumes/produces 1 token from every input/to every output.
  - No cycle: actors are fired in their topological order once to compute a set of output from a set of input.

- A set of statements without branches can be transformed into a DFG.
  - Except inputs and outputs, variables may be eliminated.
  - Ordering of operations may be relaxed.
  - Further compiling to certain target instruction set could be viewed as a scheduling of the processes/actors on a single processor.

- The functionalities of actors should match your desired level of abstraction.
- Parallelism can be exploited by firing the actors according to their levels.
Data Flow Graphs (DFG)

- Could be treated as a special SDF
  - Each actor consumes/produces 1 token from every input/to every output.
  - No cycle: actors are fired in their topological order once to compute a set of output from a set of input.

- A set of statements without branches can be transformed into a DFG.
  - Except inputs and outputs, variables may be eliminated.
  - Ordering of operations may be relaxed.
  - Further compiling to certain target instruction set could be viewed as a scheduling of the processes/actors on a single processor.

- The functionalities of actors should match your desired level of abstraction.

- Parallelism can be exploited by firing the actors according to their levels.
Data Flow Graphs (DFG)

- Could be treated as a special SDF
  - Each actor consumes/produces 1 token from every input/to every output.
  - No cycle: actors are fired in their topological order once to compute a set of output from a set of input.
- A set of statements without branches can be transformed into a DFG.
  - Except inputs and outputs, variables may be eliminated.
  - Ordering of operations may be relaxed.
  - Further compiling to certain target instruction set could be viewed as a scheduling of the processes/actors on a single processor.
- The functionalities of actors should match your desired level of abstraction.
  - Parallelism can be exploited by firing the actors according to their levels.
Data Flow Graphs (DFG)

- Could be treated as a special SDF
  - Each actor consumes/produces 1 token from every input/to every output.
  - No cycle: actors are fired in their topological order once to compute a set of output from a set of input.
- A set of statements without branches can be transformed into a DFG.
  - Except inputs and outputs, variables may be eliminated.
  - Ordering of operations may be relaxed.
  - Further compiling to certain target instruction set could be viewed as a scheduling of the processes/actors on a single processor.
- The functionalities of actors should match your desired level of abstraction.
- Parallelism can be exploited by firing the actors according to their levels.
Example Statements

inputs/outputs: u, w, y, dx, i
temporary variables: u1, u2, u3, u4, u5, u6, y1

    u1 = u *dx;
    u2 = 5 *w;
    u3 = 3 *y;
    y1 = i *dx;
    w = w +dx;
    u4 = u1*u2;
    u5 = dx*u3;
    y = y +y1;
    u6 = u -u4;
    u = u6-u5;
DFG Example

\[
\begin{align*}
    & u \\
    & \downarrow \\
    & \ast \quad \ast \quad \ast \\
    & u1 \quad u2 \quad u3 \\
    & \downarrow \quad \downarrow \quad \downarrow \\
    & - \quad + \quad * \\
    & u4 \quad dx \quad dx \\
    & \downarrow \quad \downarrow \quad \downarrow \\
    & - \quad + \\
    & u6 \quad u5 \\
    & \downarrow \quad \downarrow \\
    & u \quad w \\
    & \downarrow \quad \downarrow \\
    & y \\
    & \downarrow \\
    & y1 \\
    & \downarrow \\
    & dx
\end{align*}
\]
Outline

Data Flow Graphs

System Design Languages

C++ Review
Specify system functionality with minimum design effort by capturing models.

- Allow further automated processings/transformations, e.g. simulation/debugging, synthesis/optimization, verification.
- Natural languages are not a good choice as they are ambiguous and incomplete.

What language is ideal for system design?

Other relevant questions

- Why there are so many languages?
- Which one should I learn?
- How should I learn a particular language?
Specify system functionality with minimum design effort by capturing models.

- Allow further automated processings/transformations, e.g. simulation/debugging, synthesis/optimization, verification.
- Natural languages are not a good choice as they are ambiguous and incomplete.

What language is ideal for system design?

Other relevant questions

- Why there are so many languages?
- Which one should I learn?
- How should I learn a particular language?
Specify system functionality with minimum design effort by capturing models.
- Allow further automated processings/transformations, e.g. simulation/debugging, synthesis/optimization, verification.
- Natural languages are not a good choice as they are ambiguous and incomplete.

What language is ideal for system design?

Other relevant questions
- Why there are so many languages?
- Which one should I learn?
- How should I learn a particular language?
Typical Languages

- **C/C++/Java**
  - The core language supports sequential programs and various abstraction mechanisms, e.g. OOP, for user-defined libraries.
  - User-defined libraries cover many application domains.

- **Structural Verilog/VHDL**
  - Support mapping to hardware via a structural model, i.e. components and interconnects.
  - Primarily targeted at RTL designs.
  - Extensions support behavioral models, i.e. processes and sequential programs, though results when synthesized into hardware may be inferior.

- **Matlab**
  - For matrix computations.
  - Sequential programs are also supported, though the performance will be inferior if the computation could be written in matrix forms.
Typical Languages

- **C/C++/Java**
  - The core language supports sequential programs and various abstraction mechanisms, e.g. OOP, for user-defined libraries.
  - User-defined libraries cover many application domains.

- **Structural Verilog/VHDL**
  - Support mapping to hardware via a structural model, i.e. components and interconnects.
  - Primarily targeted at RTL designs.
  - Extensions support behavioral models, i.e. processes and sequential programs, though results when synthesized into hardware may be inferior.

- **Matlab**
  - For matrix computations.
  - Sequential programs are also supported, though the performance will be inferior if the computation could be written in matrix forms.
Typical Languages

- **C/C++/Java**
  - The core language supports sequential programs and various abstraction mechanisms, e.g. OOP, for user-defined libraries.
  - User-defined libraries cover many application domains.

- **Structural Verilog/VHDL**
  - Support mapping to hardware via a structural model, i.e. components and interconnects.
  - Primarily targeted at RTL designs.
  - Extensions support behavioral models, i.e. processes and sequential programs, though results when synthesized into hardware may be inferior.

- **Matlab**
  - For matrix computations.
  - Sequential programs are also supported, though the performance will be inferior if the computation could be written in matrix forms.
Domain Specific Languages

Languages are created to

- Solve problems in specific application domains.
- Provide abstractions so that users can adopt them for specific application domains via building their own libraries.

Application domains are distinguished by their respective models.

To learn a new language or a new library written in a language you are familiar with,

- Learn the models specific to the associated application domain.
- Learn how to capture the models using the language/library.
Domain Specific Languages

- Languages are created to
  - Solve problems in specific application domains.
  - Provide abstractions so that users can adopt them for specific application domains via building their own libraries.

- Application domains are distinguished by their respective models.
  - To learn a new language or a new library written in a language you are familiar with,
    - Learn the models specific to the associated application domain.
    - Learn how to capture the models using the language/library.
Languages are created to
- Solve problems in specific application domains.
- Provide abstractions so that users can adopt them for specific application domains via building their own libraries.

Application domains are distinguished by their respective models.

To learn a new language or a new library written in a language you are familiar with,
- Learn the models specific to the associated application domain.
- Learn how to capture the models using the language/library.
Choose a System Design Language

- Ideal system design languages should support models used to specify the whole system at all abstraction levels.
  - Can we design one?

- Practical considerations
  - Labor force: can you motivate the designers to learn this new language?
  - Legacy code: can you persuade the industry to move their code to this new language?

- Practical system design languages
  - Extend existing languages to support system designs
  - Different trade-offs leads to different choices of base languages.
  - Modern tools can support inter-operation of multiple system design languages.
Choose a System Design Language

- Ideal system design languages should support models used to specify the whole system at all abstraction levels.
  - Can we design one?

- Practical considerations
  - Labor force: can you motivate the designers to learn this new language?
  - Legacy code: can you persuade the industry to move their code to this new language?

- Practical system design languages
  - Extend existing languages to support system designs
  - Different trade-offs leads to different choices of base languages.
  - Modern tools can support inter-operation of multiple system design languages.
Choose a System Design Language

- Ideal system design languages should support models used to specify the whole system at all abstraction levels.
  - Can we design one?

- Practical considerations
  - Labor force: can you motivate the designers to learn this new language?
  - Legacy code: can you persuade the industry to move their code to this new language?

- Practical system design languages
  - Extend existing languages to support system designs
  - Different trade-offs leads to different choices of base languages.
  - Modern tools can support inter-operation of multiple system design languages.
SystemC

- A language mainly targeted for system modeling
- A front-end language based on C++
  - Due to the flexibility of C++, SystemC is entirely based on C++ – no new language constructs to learn for designers familiar with C/C++.
  - SystemC simulation environment is supported as a C++ library – legacy C/C++ code can be reused.
- Additional tools are available for verification and synthesis directly from SystemC.
SystemC

- A language mainly targeted for system modeling
- A front-end language based on C++
  - Due to the flexibility of C++, SystemC is entirely based on C++ – no new language constructs to learn for designers familiar with C/C++.
  - SystemC simulation environment is supported as a C++ library – legacy C/C++ code can be reused.
- Additional tools are available for verification and synthesis directly from SystemC.
SystemC

- A language mainly targeted for system modeling
- A front-end language based on C++
  - Due to the flexibility of C++, SystemC is entirely based on C++ – no new language constructs to learn for designers familiar with C/C++.
  - SystemC simulation environment is supported as a C++ library – legacy C/C++ code can be reused.
- Additional tools are available for verification and synthesis directly from SystemC.
SystemC Features

- VHDL/Verilog-like capabilities
  - Modules for hierarchical modeling
  - Signals and channels for communication
  - Processes for behavior modeling

- Simulation kernel to model real-world hardware behavior
- Improve simulation efficiency by leverage transaction-level modeling (TLM)
- Library to support widely applicable models, e.g. KPN
SystemC Features

- VHDL/Verilog-like capabilities
  - Modules for hierarchical modeling
  - Signals and channels for communication
  - Processes for behavior modeling
- Simulation kernel to model real-world hardware behavior
  - Improve simulation efficiency by leverage transaction-level modeling (TLM)
  - Library to support widely applicable models, e.g. KPN
SystemC Features

- VHDL/Verilog-like capabilities
  - Modules for hierarchical modeling
  - Signals and channels for communication
  - Processes for behavior modeling
- Simulation kernel to model real-world hardware behavior
- Improve simulation efficiency by leverage transaction-level modeling (TLM)
  - Library to support widely applicable models, e.g. KPN
SystemC Features

▶ VHDL/Verilog-like capabilities
  ▶ Modules for hierarchical modeling
  ▶ Signals and channels for communication
  ▶ Processes for behavior modeling
▶ Simulation kernel to model real-world hardware behavior
▶ Improve simulation efficiency by leverage transaction-level modeling (TLM)
▶ Library to support widely applicable models, e.g. KPN
SystemC Features

- VHDL/Verilog-like capabilities
  - Modules for hierarchical modeling
  - Signals and channels for communication
  - Processes for behavior modeling

- Simulation kernel to model real-world hardware behavior
  - Improve simulation efficiency by leverage transaction-level modeling (TLM)
  - Library to support widely applicable models, e.g. KPN
Outline

Data Flow Graphs

System Design Languages

C++ Review
Textbook for C++ Beginners

“Accelerated C++: Practical Programming by Example”
Koenig and Moo, Addison-Wesley, 2000.
C++ Features for Our Projects

- Sequential program
  - Standard template library (STL)
  - Object-oriented programming (OOP)
C++ Features for Our Projects

- Sequential program
- Standard template library (STL)
- Object-oriented programming (OOP)
C++ Features for Our Projects

- Sequential program
- Standard template library (STL)
- Object-oriented programming (OOP)
Statements

- Every statement is ending in ;.
- Statements may be grouped into blocks by {}.

Control constructs

- Branch: if/else
- Loop: for
Statements
  - Every statement is ending in ;.
  - Statements may be grouped into blocks by {}.

Control constructs
  - Branch: if/else
  - Loop: for
#include <stdio.h>

int main() {
    for (int i = 0; i < 10; ++i) {
        printf("%d\n", i); // display i
    }
    return 0;
}

- Use `#include` to access library functions
- Every program starts with the `main` function.
- Function: `return_type function_name(parameters) {function body}`
- `for` loop: `for (initialization; condition; increment) {for body}`
- Use `//` for comments
Sequential Program Example I

```c
#include <stdio.h>

int main() {
    for (int i = 0; i < 10; ++i) {
        printf("%d\n", i); // display i
    }
    return 0;
}
```

- Use `#include` to access library functions
- Every program starts with the `main` function.
- Function: `return_type function_name(parameters) {function body}`
- `for` loop: `for (initialization; condition; increment) {for body}`
- Use `//` for comments
#include <stdio.h>

int main() {
    for (int i = 0; i < 10; ++i) {
        printf("%d\n", i); // display i
    }
    return 0;
}

- Use #include to access library functions
- Every program starts with the main function.
- Function: return_type funtion_name(parameters) {funtion body}
- for loop: for (initialization; condition; increment) {for body}
- Use // for comments
#include <stdio.h>

int main() {
    for (int i = 0; i < 10; ++i) {
        printf("%d\n", i); // display i
    }
    return 0;
}

- Use `#include` to access library functions
- Every program starts with the `main` function.
- Function: `return_type function_name(parameters) {function body}`
- `for` loop: `for (initialization; condition; increment) {for body}`
- Use `//` for comments
#include <stdio.h>

int main() {
    for (int i = 0; i < 10; ++i) {
        printf("%d\n", i); // display i
    }
    return 0;
}

- Use `#include` to access library functions
- Every program starts with the `main` function.
- Function: `return_type funtion_name(parameters) {funtion body}`
- `for` loop: `for (initialization; condition; increment) {for body}`
- Use `//` for comments
```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int gcd(int a, int b) {
    for (; a != b;) {
        if (a > b) {
            a = a-b;
        } else {
            assert(b != a); // make sure a and b are different
            b = b-a;
        }
    }
    return a;
}
```

- **Branch:** if (condition) {if block} else {else block}
- You may omit initialization and increment for the for loop. Then it becomes the same as a while loop.
- Use assert to verify conditions
Sequential Program Example II

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int gcd(int a, int b) {
    for (; a != b;) {
        if (a > b) {
            a = a - b;
        } else {
            assert(b != a); // make sure a and b are different
            b = b - a;
        }
    }
    return a;
}
```

- **Branch:** if (condition) {if block} else {else block}
- You may omit initialization and increment for the for loop. Then it becomes the same as a while loop.
- Use assert to verify conditions
```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int gcd(int a, int b) {
    for (; a != b;) {
        if (a > b) {
            a = a-b;
        } else {
            assert(b != a); // make sure a and b are different
            b = b-a;
        }
    }
    return a;
}
```

- **Branch:** if (condition) {if block} else {else block}
- You may omit initialization and increment for the for loop. Then it becomes the same as a while loop.
- Use assert to verify conditions
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int gcd(int a, int b) {
    for (;;) {
        if (a == b) {
            return a;
        } else if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}

- Multiple if/else can be used together.
- You may omit the {} if the block contains only one statement.
```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int gcd(int a, int b) {
    for (;;) {
        if (a == b) {
            return a;
        } else if (a > b) {
            a = a-b;
        } else {
            b = b-a;
        }
    } return a;
}

▶ Multiple if/else can be used together.
▶ You may omit the {} if the block contains only one statement.
```
Standard Template Library (STL)

- Provide implementations of basic data structures and algorithms.
- Once you understand the models for those data structures and algorithms, STL becomes straight-forward.
- An example: list
  - Used to construct queues
  - What type of elements? Do we need to manually create/delete them?
  - Basic operations: traverse the list, remove/insert elements to either end.
Standard Template Library (STL)

- Provide implementations of basic data structures and algorithms.
- Once you understand the models for those data structures and algorithms, STL becomes straightforward.

- An example: list
  - Used to construct queues
  - What type of elements? Do we need to manually create/delete them?
  - Basic operations: traverse the list, remove/insert elements to either end.
Standard Template Library (STL)

► Provide implementations of basic data structures and algorithms.
► Once you understand the models for those data structures and algorithms, STL becomes straight-forward.
► An example: list
  ► Used to construct queues
  ► What type of elements? Do we need to manually create/delete them?
  ► Basic operations: traverse the list, remove/insert elements to either end.
#include <list>

int main() {
    std::list<int> queue; // use templates: a queue of integer

    queue.push_front(1); // insert 1 to the head of the list
    queue.push_back(2); // insert 2 to the tail of the list
    queue.push_back(3); // so now the list is 1,2,3
    int one = queue.front(); // get the head, which is 1
    queue.pop_front(); // remove the head, so now the list is 2,3
    int len = queue.size(); // length of the list is 2
    int two = queue.front(); // get the head, which is 2 now
    int three = queue.back(); // get the tail, which is 3 now
    queue.pop_back(); // remove the tail, so now the list is 2

    return 0;
}

- Use templates to specify element types
- Elements are copied into the list and you don’t need manage them explicitly.
- Operations are associated with meaningful member functions (accessed via .).
#include <list>

int main() {
    std::list<int> queue; // use templates: a queue of integer

    queue.push_front(1); // insert 1 to the head of the list
    queue.push_back(2); // insert 2 to the tail of the list
    queue.push_back(3); // so now the list is 1,2,3
    int one = queue.front(); // get the head, which is 1
    queue.pop_front(); // remove the head, so now the list is 2,3
    int len = queue.size(); // length of the list is 2
    int two = queue.front(); // get the head, which is 2 now
    int three = queue.back(); // get the tail, which is 3 now
    queue.pop_back(); // remove the tail, so now the list is 2

    return 0;
}

- Use templates to specify element types
- Elements are copied into the list and you don’t need manage them explicitly.
- Operations are associated with meaningful member functions (accessed via .).
#include <list>

int main() {
    std::list<int> queue; // use templates: a queue of integer

    queue.push_front(1); // insert 1 to the head of the list
    queue.push_back(2); // insert 2 to the tail of the list
    queue.push_back(3); // so now the list is 1,2,3
    int one = queue.front(); // get the head, which is 1
    queue.pop_front(); // remove the head, so now the list is 2,3
    int len = queue.size(); // length of the list is 2
    int two = queue.front(); // get the head, which is 2 now
    int three = queue.back(); // get the tail, which is 3 now
    queue.pop_back(); // remove the tail, so now the list is 2

    return 0;
}

- Use templates to specify element types
- Elements are copied into the list and you don't need manage them explicitly.
- Operations are associated with meaningful member functions (accessed via .).
Encapsulate data by using classes

- Member functions provide meaningful ways to access member data as a whole.

Polymorphism is provided at runtime via virtual functions and dynamic binding.

- To utilize dynamic binding, a derived class is defined by inheriting the base class and the virtual functions in the base class are overriden to extend/re-define their functionalities.
- Pure virtual functions (virtual functions with the =0 at the end) are used when these functions have to be overriden in a derived class.
Object-Oriented Programming (OOP)

- Encapsulate data by using classes
  - Member functions provide meaningful ways to access member data as a whole.
- Polymorphism is provided at runtime via virtual functions and dynamic binding.
  - To utilize dynamic binding, a derived class is defined by inheriting the base class and the virtual functions in the base class are overridden to extend/re-define their functionalities.
  - Pure virtual functions (virtual functions with the \(=0\) at the end) are used when these functions have to be overridden in a derived class.
class top : public sc_module // derive top from sc_module
{
public: // members and functions that can be accessed everywhere

    sc_signal<bool> clock; // a public member

    top(sc_module_name name) // constructor, or just ctor
        : sc_module(name) // initialize base class and members here
        {
    }

    int get_count() const; // a member function that won’t
                        // change the class object

protected: // members and member functions that can be accessed
        // within this class and derived classes

    int count_; // a protected member

}; // don’t forget the ;
class PE_base : public sc_module {
    ...
    void main() { // specify the functionality of PE per clock cycle
        read_input();
        execute();
        write_output();
    }
    ...
protected:
    ...
    std::list<packet> out_queue_; // output queue
    packet packet_in_; // incoming packet from the router

    virtual void read_input(); // read a packet from the the router
    virtual void execute() = 0; // abstraction of computations
    virtual void write_output(); // send a packet to the router
};

▶ PE_base::execute is a pure virtual functions that have to be overrided in a derived class.
class PE_inc : public PE_base {
    ...
    void execute();
    void fire();
    ...
};

void PE_inc::execute() {
    // fire the actor if the incoming packet is valid
    if ((packet_in_.src_x != -1) && (packet_in_.src_y != -1))
        fire();
}

void PE_inc::fire() {
    ...
    int PO_x = 0, PO_y = 0;
    packet p(x_, y_, PO_x, PO_y, packet_in_.token+1);
    ...
    out_queue_.push_back(p); // put the packet into the queue
}

▶ packet_in_ and out_queue_ are handled by PE_base::read_input and PE_base::write_output so we can just use them in the derived class.
Summary

- Sequential programs are state-based, though we may extract parallelism through its data flow.
- Languages and libraries are application specific, and thus can be learned by learning the corresponding models first.
- There is no ideal system design languages – practical system design languages have to make trade-offs.