Session 3: Wiselib Architecture

Wiselib Programming Basics: Embedded C++ and Templates

Tobias Baumgartner

Braunschweig Institute of Technology

October 13th, 2009

Outline

Outline

Outline

Having Virtual Methods

```
1  #include <stdio.h>
2
3  class BaseClass
4  {
5  public:
6    virtual int value()
7    { return 34; }
8  };
9
10 class StaticTestClass
11 {
12 public:
13   static int value( BaseClass *mc )
14   { return mc->value(); }
15 };
16
17 int main()
18 {
19   BaseClass mc;
20   int i = StaticTestClass::value( &mc );
21   printf( "Values: %d\n", i );
22 }
```

Having Templates

```
1  #include <stdio.h>
2
3  class BaseClass
4  {
5  public:
6    int value()
7    { return 34; };
8  };
9
10 template<typename Class>
11 class StaticTestClass
12 {
13 public:
14   static int value( Class *mc )
15   { return mc->value(); }
16 };
17
18 int main()
19 {
20   BaseClass mc;
21   int i = StaticTestClass<BaseClass>::value( &mc );
22   printf( "Values: %d\n", i );
23 }
```

Let's Compile!

- Virtual Method

```
1 ##################################################
2 # Virtual
3 ##################################################
4 ba-elf-g++ -g -Os -c virtual.cpp -o virtual.o
5 ba-elf-size virtual.o
6    text    data     bss     dec     hex filename
7     151       0       0     151      97 virtual.o
```

- Using the template approach

```
1 ##################################################
2 # Template
3 ##################################################
4 ba-elf-g++ -g -Os -c template.cpp -o template.o
5 ba-elf-size template.o
6    text    data     bss     dec     hex filename
7      52       0       0      52      34 template.o
```

Outline

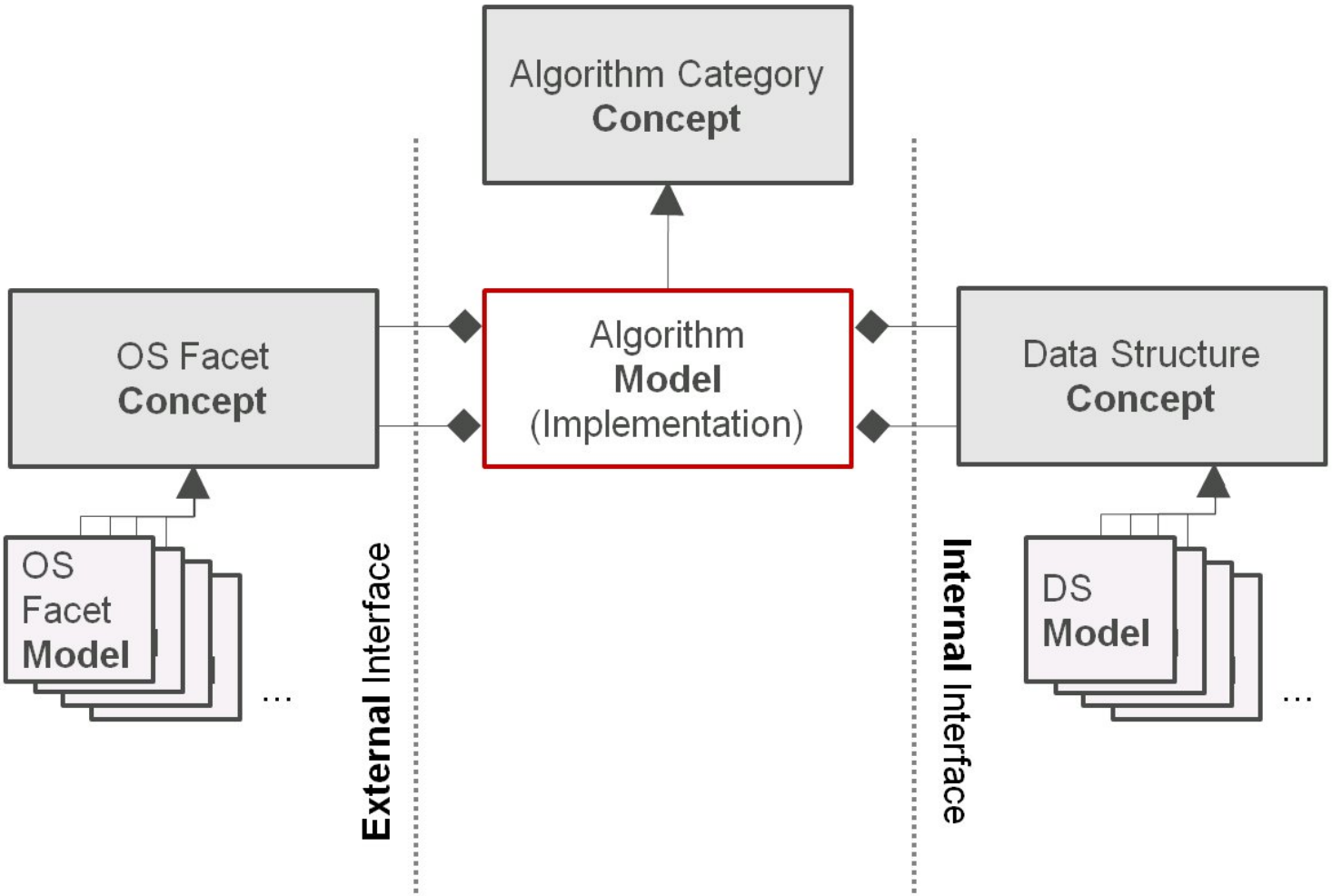
Template Based Design

```
1   class iSenseRadioModel {
2     void foo( int param );
3   }
4
5   class ShawnRadioModel {
6     void foo( int param );
7   }

8   template<typename Radio>
9   class Algorithm {
10    void bar()
11    { radio_.foo( 55 ); }
12
13    Radio radio_;
14  }

15  Algorithm<iSenseRadioModel> algorithm_isense_;
16  Algorithm<ShawnRadioModel> algorithm_shawn_;
```

Structure

- Basic design idea: **Concepts and Models**
- **Concepts** (*e.g., Radio*)
  - Documentation of a class
  - **No** written source code
  - List member functions and provided types
  - Interface description
- **Models** (*e.g., iSenseRadioModel*)
  - Implementation of a model
  - Must implement each listed meber function and type
  - There can be multiple models per concept
  - One model can implememt multiple concepts
- **Usage** (*e.g., Algorithm*)
  - Model is expected as template parameter
  - In implementation, rely on concept description
  - Compiler can resolve everything at compile-time

Wiselib Architecture



Outline

Outline

Overview

Facets

Os	Radio	Timer
typedef Radio typedef Timer typedef Clock typedef size_t	enable() disable() send() reg_recv_cb() unreg_recv_cb() id()	set_timer()

Models

Structure

- **Connection** between Algorithms and OS
- **OS Facets**
  - Os Facet
  - Radio Facet
  - Timer Facet
  - ...
- For **each** supported OS at least **one model** per **facet**
  - iSenseOsModel
  - ContikiRadioModel
  - ...
- Possible to provide muliple models per facet
- Specifics due to underlying Os: C, nesC, C++
  - Callback registration (C functions vs. Member functions)
  - We **do not** want to have instances of any Os Facet Model

Os Facet Details

- **Static Member Functions**

```
concept Radio {
  ...
  static inline void enable( Os *os );
  ...
}
```

- **OS Pointer** contains information about the underlying Os
  - In iSense it is `isense::Os`
  - In Shawn it is the corresponding `shawn::Processor`
  - In Contiki it is just `void`

```
static inline void enable( Os *os )
{
  os->radio().enable();
}
```

Example: Radio Concept

```
1  concept Radio {
2      typedef ... OsModel;
3      typedef ... Os;
4      typedef ... node_id_t;
5      typedef ... data_t;
6      typedef ... size_t;
7      typedef ... message_id_t;
8
9      enum SpecialNodeIds { BROADCAST_ADDRESS=..., NULL_NODE_ID=... };
10     enum Restrictions { MAX_MESSAGE_LENGTH = ... };
11
12     static inline int
13     send(Os *os, node_id_t id, size_t len, const data_t *data);
14
15     static int enable( Os *os );
16     static int disable( Os *os );
17     static node_id_t id( Os *os );
18
19     template<class T, void (T::*TMethod)(node_id_t, size_t, data_t*)>
20     static int reg_recv_callback( Os *os, T *obj_pnt );
21     static void unreg_recv_callback( Os *os, int idx );
22 };
```

Example: Use Radio in Algorithm

```
1  template <...,
2            typename Radio_P,
3            ...>
4  class MyAlgorithm
5  {
6    typedef typename Radio_P Radio;
7    ...
8    foo( void );
9    ...
10 }
11 // ----------------------------------------------------------------------
12 template <...,
13           typename Radio_P,
14           ...>
15 void
16 MyAlgorithm<..., Radio_P, ...>::
17 foo( void )
18 {
19   ...
20   Radio::enable( os() );
21   ...
22 }
```

Outline

Example: Radio Usage

- Enable/Disable

```
1 Radio::enable( os() );
2 Radio::disable( os() );
```

- Sending a message

```
1 char message[] = "Hello World!";
2 Radio::node_id_t dest = Radio::BROADCAST_ADDRESS;
3 Radio::send( os(), dest, sizeof(msg), (block_data_t*)msg );
```

- Register receive method (within a template)

```
1 Radio::template reg_recv_callback<MyClass, &MyClass::receive>
2     ( os(), this );
3 ...
4 template<...>
5 void MyClass<...>::receive( node_it_t, size_t, block_data_t )
6 { ... }
```

Note that the name of the method (here: `receive`) does **not** matter! Only the signature must be correct. Hence you could register different methods at different radios, for example.

Example: Timer Usage

- Set timer within a template

```
1 Timer::template set_timer<MyClass, &MyClass::timer_elapsed>(
2                                   os(), time_in_ms, this, userdata );
3 ...
4 template<...>
5 void MyClass<...>::timer_elapsed( void *userdata )
6 { ... }
```

- Set timer outside a template

```
1 Timer::set_timer<MyClass, &MyClass::timer_elapsed>(
2                                   os(), time_in_ms, this, userdata );
3 ...
4 void MyClass<...>::timer_elapsed( void *userdata )
5 { ... }
```

Outline

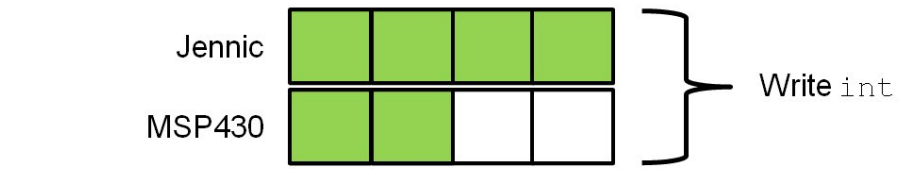
Structure

- Data structures used by algorithms
  - Various implementations for different demands
  - Static vs Dynamic
- pSTL: picoSTL for static containers
  - Iterator
  - Vector
  - List
  - Priority Queue
  - Pair
  - ...

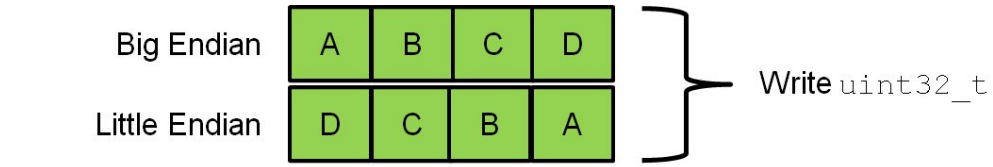
Outline

Problems in Heterogeneous Systems

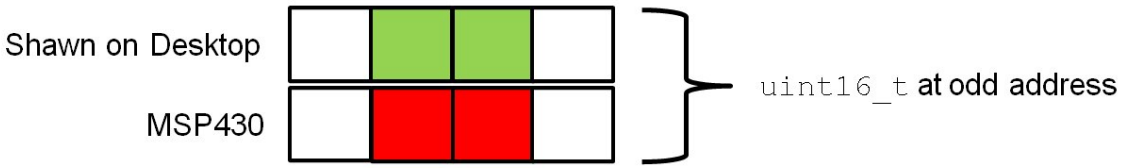
1. Bit Width



2. Byte Order



3. Alignment



Serialization

- Templated Serialization provided by Wiselib

```
1  template<typename OsModel_P, typename Data_P, typename Type_P>
2  inline Type_P read( Data_P *target )
3  {
4    return Serialization<OsModel_P, Data_P, Type_P>::
5              read( target );
6  }
7
8  template<typename OsModel_P, typename Data_P, typename Type_P>
9  inline typename OsModel_P::size_t write(Data_P *target,
10                                         Type_P& value)
11 {
12   return Serialization<OsModel_P, BlockData_P, Type_P>::
13             write( target, value );
14 }
```

General Approach

- Provide basic implementation

```
1  template<typename OsModel_P, typename Data_P, typename Type_P>
2  class Serialization
3  {
4    public:
5      typedef OsModel_P OsModel;
6      typedef Data_P BlockData;
7      typedef Type_P Type;
8
9      typedef typename OsModel::size_t size_t;
10     // ------------------------------------------------------------------
11     static inline size_t write( BlockData *target, Type& value )
12     {
13       *((Type*)target) = value;
14       return sizeof(Type);
15     }
16     ...
```

Spezialization

- Specialize for Platforms or Types

```
1  template<typename OsModel_P, typename BlockData_P>
2  class Serialization<OsModel_P, BlockData_P, uint16_t>
3  {
4    public:
5      typedef OsModel_P OsModel;
6      typedef BlockData_P BlockData;
7      typedef uint16_t Type;
8
9      typedef typename OsModel::size_t size_t;
10     //--------------------------------------------------------------------------
11     static inline size_t write( BlockData *target, Type& value )
12     {
13       *target = (value >> 8) & 0xff;
14       *(target + 1) = value & 0xff;
15       return sizeof(uint16_t);
16     }
17     ...
```

Result

- Can be specialized for **each system** and **each data type**
- Data Access in Messages

```
1 inline uint8_t msg_id()
2 { return read<OsModel, uint8_t>(buffer); }
3
4 inline void set_msg_id( uint8_t id )
5 { write<OsModel, block_data_t, uint8_t>(buffer, id); }
```

Outline

Overview

- Heart of Wiselib
- Provide different algorithm categories
  - Routing
  - Crypto
  - Clustering
  - Localization
  - ...
- One **Concept** for each category
- Multiple implementations (**models**) per concept
  - DsdvRouting
  - HeedClustering
  - ...

Advantages and Special Design Issues

- Stackability
  - Stack one algorithm into another
  - E.g., easily combine Crypto with Routing to get Secure Routing
- Interchangeability
  - Exchange one algorithm within a second or two
  - E.g., change `typedefs` to use AODV instead of DSDV
- Cross-Layer Design
  - One model implements multiple concepts
  - E.g., clusterbased routing

Outline

Wiselib Demands

- Different demands: *Application Developer* and *Algorithm Developer*
- Application Developer
  - Integrate Wiselib in own applications
  - Use Wiselib algorithms Running on all supported platforms
  - Thoroughly tested
- Algorithm Developer
  - Integration of own implementations
    - May be tested only on available platform
    - May **not** be thoroughly tested
  - Add new concepts
    - Must be discussed with Wiselib contributors
    - May change due to iterative development process
- Solution: Distributions Stable, Testing, and Incubation

Distributions

- Incubation
  - Implementations for arbitrary architectures (e.g., Contiki)
  - Not required to use Wiselib API
  - Beneficial when porting an algorithm to Wiselib
  - `wiselib/trunk/wiselib.incubation`
- Testing
  - Implemented against Wiselib API
  - Algorithms may only be tested on some target platforms
  - New concepts that may change
  - Release Early, Release Often
  - `wiselib/trunk/wiselib.testing`
- Stable
  - Tested on all supported platforms
  - Concepts may not change anymore
  - `wiselib/trunk/wiselib.stable`

Outline

Exercise for this Session

- Write a simple flooding algorithm for the Wiselib.
  - Any node is able to send a message
  - That message should be sent to each node in the network
- The algorithm should implement the `Routing Concept` (concept given on next slide)
- Template for algorithm and message class is given
- Properties that the algorithm should have:
  - Get at least `Os` and `Radio` as template parameter
  - Do not send the same message twice
  - Reliability does not matter (do not care about message loss, unless you want to... :)
- Download sample files:

  `http://www.ibr.cs.tu-bs.de/winterschool/session3-template.tgz`

Routing Concept

```
1  concept Routing
2  {
3      typedef ... OsModel;
4      typedef ... RoutingTable;
5      typedef ... Radio;
6      typedef ... Os;                ///< Can just be OsModel::Os
7      typedef ... self_type;
8      typedef ... node_id_t;
9      typedef ... size_t;
10     typedef ... block_data_t;
11
12     void enable( void );
13     void disable( void );
14
15     void send( node_id_t receiver, size_t len, block_data_t *data );
16
17     template<class T, void (T::*TMethod)(node_id_t, size_t, block_data
18         int reg_recv_callback( T *obj_pnt );
19     void unreg_recv_callback( int idx );
20
21     void set_os( Os* os );
22     Os* os();
23 };
```

Hints

- Use the provided Template
  (here: *template* is meant as sample or draft)
- Note that the `Flooding` is derived from `RoutingBase`:
  (un)`reg_recv_callback` must **not** be implemented
- What should be done in `enable`:
  - Enable radio
  - Register at radio as receiver
- What should be done in `disable`:
  - Disable radio
  - Unregister at radio as receiver