Konfigurierbare Systemsoftware (KSS)

VL 3 – Aspect-Oriented Programming (AOP)

Daniel Lohmann
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität
Erlangen-Nürnberg
SS 16 – 2016-04-25
http://www4.informatik.uni-erlangen.de/Lehre/SS16/V_KSS

Implementation Techniques: Classification

- Decompositional Approaches
  - Text-based filtering (untyped)
  - Preprocessors

- Compositional Approaches
  - Language-based composition mechanisms (typed)
  - OOP, AOP, Templates

- Generative Approaches
  - Metamodel-based generation of components (typed)
  - MDD, C++ TMP, generators

About this Lecture

Agenda

3.1 Motivation: Separation of Concerns
3.2 Tutorial: AspectC++
3.3 Summary and Outlook
3.4 References
Agenda

3.1 Motivation: Separation of Concerns
   Example: I4WeatherMon
   Example: eCos
AOP
3.2 Tutorial: AspectC++
3.3 Summary and Outlook
3.4 References

Case Study eCos [4]

- The embedded Configurable operating system
  - Operating system for embedded applications
  - Open source, maintained by eCosCentric Inc.
  - Many 16-bit and 32-bit platforms supported
  - Broadly accepted real-world system
- More than 750 configuration options (kernel)
  - Feature-based selection
  - Preprocessor-based implementation

Static Configurability with the CPP?

I4WeatherMon (CPP): Implementation (Excerpt)

Mutex options:
- PROTOCOL
- CEILING
- INHERIT
- DYNAMIC

Kernel policies:
- Tracing
- Instrumentation
- Synchronization

I4WeatherMon example from last lecture

Symbolic preprocessing can be used to separate different concerns and allow for configurability in embedded systems.
AOP is about modularizing crosscutting concerns. Examples: tracing, synchronization, buffering, error handling, constraint checks, ...

Kernel policies:

- Tracing
- Instrumentation
- Synchronization

 Mutex options:
- PROTOCOL
- CEILING
- INHERIT
- DYNAMIC

Aspect-Oriented Programming (AOP)

AOP is about modularizing crosscutting concerns.

Examples: tracing, synchronization, security, buffering, error handling, constraint checks, ...
AOP: The Basic Idea

Separation of what from where:

- **Join Points** ➞ **where**
  - positions in the static structure or dynamic control flow (event)
  - given declaratively by pointcut expressions

- **Advice** ➞ **what**
  - additional elements (members, ...) to introduce at join points
  - additional behavior (code) to superimpose at join points

Implementation of Crosscutting Concerns with AOP

```c
aspect int_sync {
  pointcut sync() = execution(...) // kernel calls to sync
  | construction(...)
  | destruction(...);

  // advise kernel code to invoke lock() and unlock()
  advice sync() : before() {
    Cyg_Scheduler::lock();
  }
  advice sync() : after() {
    Cyg_Scheduler::unlock();
  }

  // In eCos, a new thread always starts with a lock value of 0
  advice execution("%Cyg_HardwareThread::thread_entry(...)") : before() {
    Cyg_Scheduler::zero_sched_lock();
  }
};
```

Static Configurability with the CPP?

Crosscutting Concerns
Can we do better with aspects?

Result
after refactoring into aspects [4]
AspectC++

- AspectC++ is an AOP language extension for C++
  - superset of ISO C++ 98 [1]
  - every C++ program is also an AspectC++ program
- additionally supports AOP concepts
- Technical approach: source-to-source transformation
  - ac++ weaver transforms AspectC++ code into C++ code
  - resulting C++ code can be compiled with any standard-compliant compiler (especially gcc)
  - ag++ weaver wrapper works as replacement for g++ in makefiles
- Language and weaver are open source (GPL2)

http://www.aspectc.org

Scenario: A Simple Queue

namespace util {
  class Item {
    friend class Queue;
    Item* next;
  public:
    Item() : next(0){}
  }
  class Queue {
    Item* first;
    Item* last;
  public:
    Queue() : first(0), last(0) {
      printf( "\n   > Queue::enqueue()
" );
      if ( last ) {
        last->next = item;
        last = item;
      } else
        last = first = item;
      printf( "\n   < Queue::enqueue()
" );
    }
    Item* dequeue() {
      printf("\n   > Queue::dequeue()
" );
      Item* res = first;
      if ( first == last )
        first = last = 0;
      else
        first = first->next;
      printf("\n   < Queue::dequeue()
" );
      return res;
    }
  } // class Queue
} // namespace util

Agenda

3.1 Motivation: Separation of Concerns
3.2 Tutorial: AspectC++
  Example Scenario
  First Steps And Language Overview
  Advanced Concepts
  Weaver Transformations
  Further Examples
3.3 Summary and Outlook
3.4 References
Scenario: The Problem
Please extend the Queue class by an element counter!
I want Queue to throw exceptions!
Queue should be thread-safe!
Various users of Queue demand extensions:

Problem Summary
The component code is “polluted” with code for several logically independent concerns, thus it is ...

- hard to write the code
  - many different things have to be considered simultaneously
- hard to read the code
  - many things are going on at the same time
- hard to maintain and evolve the code
  - the implementation of concerns such as locking is scattered over the entire source base (a “crosscutting concern”)
- hard to configure at compile time
  - the users get a “one fits all” queue class
Goal: A configurable Queue

Configuring with the Preprocessor?

```cpp
class Queue {
  Item *first, *last;
#ifdef COUNTING_ASPECT
  int counter;
#endif
#ifdef LOCKING_ASPECT
  os::Mutex lock;
#endif
public:
  Queue () : first(0), last(0) {
#ifdef COUNTING_ASPECT
    counter = 0;
#endif
  }
  void enqueue(Item* item) {
#ifdef LOCKING_ASPECT
    lock.enter();
    try {
#endif
#ifdef ERRORHANDLING_ASPECT
    if (item == 0)
      throw QueueInvalidItemError();
#endif
    if (last) {
      last->next = item;
      last = item;
    } else
    { last = first = item; }
#ifdef COUNTING_ASPECT
    ++counter;
#endif
#ifdef LOCKING_ASPECT
    }
    catch (...) {
      lock.leave();
      throw;
    }
    lock.leave();
  }
  Item* dequeue() {
    Item* res;
    #ifdef LOCKING_ASPECT
      lock.enter();
      try {
    #endif
    res = first;
    if (first == last)
      first = last = 0;
    else
    first = first->next;
    #ifdef COUNTING_ASPECT
    if (counter > 0) --counter;
    #endif
    #ifdef ERRORHANDLING_ASPECT
    if (res == 0)
      throw QueueEmptyError();
    #endif
    #ifdef LOCKING_ASPECT
      }
    catch (...) {
      lock.leave();
      throw;
    }
    lock.leave();
    return res;
  }
#ifdef COUNTING_ASPECT
  int count() { return counter; }
#endif
}; // class Queue
```

Queue: Element Counting

Queue: Demanded Extensions

I. Element counting

II. Error handling (signaling of errors by exceptions)

III. Thread safety (synchronization by mutex variables)
Queue: Element Counting

Element counting: The Idea

➢ Increment a counter variable after each execution of `util::Queue::enqueue()`
➢ Decrement it after each execution of `util::Queue::dequeue()`

ElementCounter1

```cpp
aspect ElementCounter {
  int counter;
  ElementCounter() {
    counter = 0;
  }
  advice execution("% util::Queue::enqueue(...)") : after() {
    ++counter;
    printf( " Aspect ElementCounter: # of elements = %d\n", counter );
  }
  advice execution("% util::Queue::dequeue(...)") : after() {
    if( counter > 0 ) --counter;
    printf( " Aspect ElementCounter: # of elements = %d\n", counter );
  }
};
```

We introduced a new `aspect` named `ElementCounter`. An aspect starts with the keyword `aspect` and is syntactically much like a class.

Like a class, an aspect can define data members, constructors and so on.
We give after advice (= some crosscutting code to be executed after certain control flow positions).

Aspect member elements can be accessed from within the advice body.

This pointcut expression denotes where the advice should be given. (After execution of methods that match the pattern)

Aspect member elements can be accessed from within the advice body.
The aspect is not the ideal place to store the counter, because it is shared between all Queue instances.

Ideally, counter becomes a member of Queue.

In the next step, we

- move counter into Queue by introduction
- expose context about the aspect invocation to access the current Queue instance.

Introduces a slice of members into all classes denoted by the pointcut "util::Queue".

We introduce a private counter element and a public method to read it.
Aspect ElementCounter {
  advice "util::Queue" : slice class {
    int counter;
    
    +queue, counter;
    printf(" Aspect ElementCounter: # of elements = \%d\n", queue.count());
  }
  
  after( util::Queue& queue ) {
    queue.counter = 0;
  }
};

By giving construction advice we ensure that counter gets initialized.

int main() {
  util::Queue queue;
  printf("main(): Queue contains %d items\n", queue.count());
  queue.enqueue(new util::Item);
  printf("main(): enqueueing some items\n");
  queue.enqueue(new util::Item);
  printf("main(): Queue contains %d items\n", queue.count());
  printf("main(): dequeueing one items\n");
  util::Item item;
  item = queue.dequeue();
  printf("main(): Queue contains %d items\n", queue.count());
  item.print();
};
Queue: Element Counting

ElementCounter2 - Result

```c
int main() {
  util::Queue queue;
  printf("main(): Queue contains %d items\n", queue.count());
  queue.enqueue(new util::Item);
  printf("main(): enqueueing one items\n");
  printf("main(): Queue contains %d items\n", queue.count());
  util::Item* item = queue.dequeue();
  printf("main(): dequeueing one items\n");
  printf("main(): Queue contains %d items\n", queue.count());
  queue.enqueue(00320FD0);
  printf("main(): enqueueing some items\n");
  printf("main(): Queue contains %d items\n", queue.count());
  Aspect ElementCounter: # of elements = 1
  Aspect ElementCounter: # of elements = 2
  Aspect ElementCounter: # of elements = 1
}
```

You have seen...

- the most important concepts of AspectC++
  - Aspects are introduced with the keyword `aspect`
  - They are much like a class, may contain methods, data members, types, inner classes, etc.
  - Additionally, aspects can give `advice` to be woven in at certain positions (joinpoints). Advice can be given to:
    - Functions/Methods/Constructors: code to execute (code advice)
    - Classes or structs: new elements (introductions)
  - Joinpoints are described by `pointcut expressions`

- We will now take a closer look at some of them

AspectC++ Language Elements

Joinpoints

➢ A joinpoint denotes a position to give advice
  - Code joinpoint
    a point in the control flow of a running program, e.g.
    - `execution` of a function
    - `call` of a function
  - Name joinpoint
    • a named C++ program entity (identifier)
    • class, function, method, type, namespace

➢ Joinpoints are given by `pointcut expressions`
  - a pointcut expression describes a set of joinpoints
Pointcut Expressions

- **Pointcut expressions** are made from...
  - match expressions, e.g. `<% util::queue::enqueue(...)">`
    - are matched against C++ programm entities ➔ name joinpoints
    - support wildcards
  - pointcut functions, e.g. execution(...), call(...), that(...)
    - execution: all points in the control flow, where a function is about to be executed ➔ code joinpoints
    - call: all points in the control flow, where a function is about to be called ➔ code joinpoints

- **Pointcut functions** can be combined into expressions
  - using logical connectors: &&, ||, !
  - Example: call("% util::Queue::enqueue(...)") && within("% main(...)")

Advice to functions

- **before advice**
  - Advice code is executed before the original code
  - Advice may read/modify parameter values
- **after advice**
  - Advice code is executed after the original code
  - Advice may read/modify return value
- **around advice**
  - Advice code is executed instead of the original code
  - Original code may be called explicitly: tjp->proceed()

Introductions

- A slice of additional methods, types, etc. is added to the class
- Can be used to extend the interface of a class
### Introductions

```cpp
class ClassA {
public:
  void foo() {
    printf("ClassA::foo()\n");
  }
};
```

### Queue: Error Handling

#### Errorhandling: The Idea

- We want to check the following constraints:
  - `enqueue()` is never called with a NULL item
  - `dequeue()` is never called on an empty queue
- In case of an error an exception should be thrown
- To implement this, we need access to ...
  - the parameter passed to `enqueue()`
  - the return value returned by `dequeue()`
  ... from within the advice

#### ErrorException

```cpp
namespace util {
  struct QueueInvalidItemError {};
  struct QueueEmptyError {};
}
```
ErrorException – Lessons Learned

You have seen how to...

➢ use different types of advice
  - before advice
  - after advice

➢ expose context in the advice body
  - by using `args` to read/modify parameter values
  - by using `result` to read/modify the return value
Queue: Thread Synchronization

Queue: Demanded Extensions

I. Element counting

II. Errorhandling
(signaling of errors by exceptions)

III. Thread safety
(synchronization by mutex variables)

Queue should be thread-safe!

Thread Safety: The Idea

➢ Protect enqueue() and dequeue() by a mutex object
➢ To implement this, we need to
  – introduce a mutex variable into class Queue
  – lock the mutex before the execution of enqueue() / dequeue()
  – unlock the mutex after execution of enqueue() / dequeue()
➢ The aspect implementation should be exception safe!
  – in case of an exception, pending after advice is not called
  – solution: use around advice

aspect LockingMutex {
  advice "util::Queue" : slice class { os::Mutex lock; };
  pointcut sync_methods() = "% util::Queue::%queue(...)";
  advice execution(sync_methods()) && that(queue) :
    around( util::Queue& queue ) {
      queue.lock.enter();
      try {
        tjp->proceed();
      } catch(...) {
        queue.lock.leave();
        throw;
      }
      queue.lock.leave();
    }
};

We introduce a mutex member into class Queue

lockingMutex.ah

lockingMutex - Elements

aspect LockingMutex {
  advice "util::Queue" : slice class { os::Mutex lock; };
  pointcut sync_methods() = "% util::Queue::%queue(...)";
  advice execution(sync_methods()) && that(queue) :
    around( util::Queue& queue ) {
      queue.lock.enter();
      try {
        tjp->proceed();
      } catch(...) {
        queue.lock.leave();
        throw;
      }
      queue.lock.leave();
    }
};
LockingMutex - Elements

aspect LockingMutex {
  advice "util::Queue" : slice class { os::Mutex lock; };
  pointcut sync_methods() = "% util::Queue::%queue(...)";
  advice execution(sync_methods()) & that(queue)
    : around( util::Queue& queue ) { 
      queue.lock.enter();
      try {
        tjp->proceed();
      }
      catch(...) { 
        queue.lock.leave();
        throw;
      }
    }
    queue.lock.leave();
  }
};

Pointcuts can be named. `sync_methods` describes all methods that have to be synchronized by the mutex.

LockingMutex.ah

aspect LockingMutex {
  advice "util::Queue" : slice class { os::Mutex lock; };
  pointcut sync_methods() = "% util::Queue::%queue(...)";
  advice execution(sync_methods()) & that(queue)
    : around( util::Queue& queue ) { 
      queue.lock.enter();
      try {
        tjp->proceed();
      }
      catch(...) { 
        queue.lock.leave();
        throw;
      }
    }
    queue.lock.leave();
};

sync_methods is used to give around advice to the execution of the methods.

LockingMutex.ah

Queue: Thread Synchronization

LockingMutex - Elements

aspect LockingMutex {
  advice "util::Queue" : slice class { os::Mutex lock; };
  pointcut sync_methods() = "% util::Queue::%queue(...)";
  advice execution(sync_methods()) & that(queue)
    : around( util::Queue& queue ) { 
      queue.lock.enter();
      try {
        tjp->proceed();
      }
      catch(...) { 
        queue.lock.leave();
        throw;
      }
    }
    queue.lock.leave();
};

By calling tjp->proceed() the original method is executed.

LockingMutex.ah

LockingMutex – Lessons Learned

You have seen how to...

➢ use named pointcuts
  - to increase readability of pointcut expressions
  - to reuse pointcut expressions
➢ use around advice
  - to deal with exception safety
  - to explicit invoke (or don't invoke) the original code by calling tjp->proceed()
➢ use wildcards in match expressions
  - "% util::Queue::%queue(...)" matches both enqueue() and dequeue()
Queue: IRQ Synchronization

Queue: A new Requirement

I. Element counting

II. Error handling
   (signaling of errors by exceptions)

III. Thread safety
   (synchronization by mutex variables)

IV. Interrupt safety
   (synchronization on interrupt level)

We need Queue to be synchronized on interrupt level!

Interrupt Safety: The Idea

➢ Scenario
– Queue is used to transport objects between kernel code (interrupt handlers) and application code
– If application code accesses the queue, interrupts must be disabled first
– If kernel code accesses the queue, interrupts must not be disabled

➢ To implement this, we need to distinguish
   – if the call is made from kernel code, or
   – if the call is made from application code

aspect LockingIRQ1

pointcut sync_methods() = "% util::Queue::%queue(...)";
pointcut kernel_code() = "% kernel::%(...)";

advice call(sync_methods()) && !within(kernel_code()) : around()
{
  os::disable_int();
  try{
    tjp->proceed();
  }
  catch(...) {
    os::enable_int();
    throw;
  }
  os::enable_int();
}

LockingIRQ1.ah

We define two pointcuts. One for the methods to be synchronized and one for all kernel functions
LockingIRQ1 – Elements

aspect LockingIRQ {
  pointcut sync_methods() = "% util::Queue::%queue(...)";
  pointcut kernel_code() = "% kernel::%(...)";
  advice call(sync_methods()) & within(kernel_code()) : around() {
    os::disable_int();
    try {
      tjp->proceed();
    } catch(...) {
      os::enable_int();
    }
  }
};

LockingIRQ1.ah

This pointcut expression matches any call to a sync_method that is not done from kernel_code.

LockingIRQ1 – Problem

main.cc
util::Queue queue;
void do_something() {
  printf("do_something()\n");
  queue.enqueue(new util::Item);
  kernel::irq_handler();
  do_something();
  queue.dequeue();
}

main() {
  printf("kernel::irq_handler()\n");
  namespace kernel {
    void irq_handler() {
      printf("kernel::irq_handler()\n");
      queue.enqueue(new util::Item);
      kernel::irq_handler(); // irq
      printf("back in main()\n");
      queue.dequeue();
    }
  }
}

The pointcut within(kernel_code) does not match any indirect calls to sync_methods.

LockingIRQ1 – Result

main.cc
util::Queue queue;
void do_something() {
  printf("do_something()\n");
  queue.enqueue(new util::Item);
  os::disable_int();
  try {
    tjp->proceed();
  } catch(...) {
    os::enable_int();
  }
  os::disable_int();
  queue.enqueue(new util::Item);
  back in main()
  os::enable_int();
  > Queue::enqueue() returning 00320FD0
  os::enable_int();
  os::disable_int();
  main()
  os::disable_int()}

LockingIRQ2

aspect LockingIRQ {
  pointcut sync_methods() = "% util::Queue::%queue(...)";
  pointcut kernel_code() = "% kernel::%(...)";
  advice execution(sync_methods()) & within(kernel_code()) : around() {
    os::disable_int();
    try {
      tjp->proceed();
    } catch(...) {
      os::enable_int();
      throw;
    }
    os::enable_int();
    > Queue::enqueue() returning 00320FD0
    os::enable_int();
  }
};

LockingIRQ2.ah

Using the cflow pointcut function

Solution

Using the cflow pointcut function
LockingIRQ – Lessons Learned

You have seen how to ...

➢ restrict advice invocation to a specific calling context

➢ use the within(...) and cflow(...) pointcut functions
  - within is evaluated at compile time and returns all code
    joinpoints of a class' or namespaces lexical scope
  - cflow is evaluated at runtime and returns all joinpoints
    where the control flow is below a specific code joinpoint

A First Summary

AspectC++: A First Summary

➢ The Queue example has presented the most important features of the AspectC++ language
  - aspect, advice, joinpoint, pointcut expression, pointcut function, ...

➢ Additionally, AspectC++ provides some more advanced concepts and features
  - to increase the expressive power of aspectual code
  - to write broadly reusable aspects
  - to deal with aspect interdependence and ordering

➢ In the following, we give a short overview on these advanced language elements

Queue: IRQ Synchronization

LockingIRQ2 – Elements

```cpp
aspect LockingIRQ {
  pointcut sync_methods() = "% util::Queue::%queue(...)";
  pointcut kernel_code() = "% kernel::%(...);";
  advice execution(sync_methods()) & if cflow(execution(kernel_code())) : around {
    os::disable_int();
    try {
      tjp->proceed();
    } catch(...) {
      os::disable_int();
      throw;
    }
    os::enable_int();
  };
}
```

This pointcut expression matches the execution of sync_methods if no kernel_code is on the call stack. cflow checks the call stack (control flow) at runtime.

Queue: IRQ Synchronization

LockingIRQ2 – Result

```cpp
main.cc
util::Queue queue;
void do_something() {
  printf("do_something\n");
  queue.enqueue(00320FD0)
  > Queue::enqueue()
  < Queue::enqueue() returning 00320FD0
os::disable_int();
<Output>
```

A First Summary

AspectC++: A First Summary

➢ The Queue example has presented the most important features of the AspectC++ language
  - aspect, advice, joinpoint, pointcut expression, pointcut function, ...

➢ Additionally, AspectC++ provides some more advanced concepts and features
  - to increase the expressive power of aspectual code
  - to write broadly reusable aspects
  - to deal with aspect interdependence and ordering

➢ In the following, we give a short overview on these advanced language elements

Queue: IRQ Synchronization

LockingIRQ2 – Elements

```cpp
aspect LockingIRQ {
  pointcut sync_methods() = "% util::Queue::%queue(...)";
  pointcut kernel_code() = "% kernel::%(...);";
  advice execution(sync_methods()) & if cflow(execution(kernel_code())) : around {
    os::disable_int();
    try {
      tjp->proceed();
    } catch(...) {
      os::disable_int();
      throw;
    }
    os::enable_int();
  };
}
```

This pointcut expression matches the execution of sync_methods if no kernel_code is on the call stack. cflow checks the call stack (control flow) at runtime.
AspectC++: Advanced Concepts

➢ Join Point API
  • provides a uniform interface to the aspect invocation context, both at runtime and compile-time

➢ Abstract Aspects and Aspect Inheritance
  • comparable to class inheritance, aspect inheritance allows to reuse parts of an aspect and overwrite other parts

➢ Generic Advice
  • exploits static type information in advice code

➢ Aspect Ordering
  • allows to specify the invocation order of multiple aspects

➢ Aspect Instantiation
  • allows to implement user-defined aspect instantiation models

The Join Point API (Excerpt)

Types (compile-time)
  // object type (initiator)
  That
  // object type (receiver)
  Target
  // result type of the affected function
  Result
  // type of the i'th argument of the affected function
  Arg<i>::Type
  Arg<i>::ReferredType

Values (runtime)
  // pointer to the object initiating a call
  That* that()
  // pointer to the object that is target of a call
  Target* target()
  // pointer to the result value
  Result* result()
  // typed pointer the i'th argument value of a function call
  Arg<i>::Type* arg(i)
  Arg<i>::ReferredType* arg(i)
  // pointer the i'th argument value of a function call
  void* arg(i)

Consts (compile-time)
  // number of arguments
  ARGS
  // unique numeric identifier for this join point
  JPID
  // numeric identifier for the type of this join point
  JPID
  // function (with 0 <= i < ARGS)
  function((0 <= i < ARGS), Arg<i>::Type,
  Arg<i>::ReferredType)

The Joinpoint API

➢ Inside an advice body, the current joinpoint context is available via the implicitly passed tjp variable:

  advice ... {
    struct JoinPoint {
      ...
      *tjp: // implicitly available in advice code
      ...
    }
    tjp->proceed()
  }

➢ You have already seen how to use tjp, to ...
  • execute the original code in around advice with tjp->proceed()

➢ The joinpoint API provides a rich interface
  • to expose context independently of the aspect target
  • this is especially useful in writing reusable aspect code
Abstract Aspects and Inheritance

- Aspects can inherit from other aspects...
  - Reuse aspect definitions
  - Override methods and pointcuts
- Pointcuts can be pure virtual
  - Postpone the concrete definition to derived aspects
  - An aspect with a pure virtual pointcut is called an abstract aspect
- Common usage: Reusable aspect implementations
  - Abstract aspect defines advice code, but pure virtual pointcuts
  - Aspect code uses the joinpoint API to expose context
  - Concrete aspect inherits the advice code and overrides pointcuts

```cpp
#include "mutex.h"
aspect LockingA {
  pointcut virtual sync_classes() = 0;
  pointcut virtual sync_methods() = 0;
  advice sync_classes() : slice class {
    os::Mutex lock;
  }
  advice execution(sync_methods()) : around() {
    tjp->that()->lock.enter();
    try {
      tjp->proceed();
    } catch(...) {
      tjp->that()->lock.leave();
    }
  }
};
```

The abstract locking aspect declares two pure virtual pointcuts and uses the joinpoint API for a context-independent advice implementation.

```cpp
aspect LockingQueue : public LockingA {
  pointcut sync_classes() = "util::Queue":
  pointcut sync_methods() = "% util::Queue::%queue(...)";
};
```

The concrete locking aspect derives from the abstract aspect and overrides the pointcuts.
Advanced Concepts

Generic Advice

Uses static JP-specific type information in advice code
- in combination with C++ overloading
- to instantiate C++ templates and template meta-programs

Resolves to the statically typed return value
- no runtime type checks are needed
- unhandled types are detected at compile-time
- functions can be inlined

aspect TraceService {
  advice call(...) : after() {
    ... cout << *tjp->result();
  }
};

... operator <<(..., int)
... operator <<(..., long)
... operator <<(..., bool)
... operator <<(..., Foo)

Aspect Ordering

➢ Aspects should be independent of other aspects
  - However, sometimes inter-aspect dependencies are unavoidable
  - Example: Locking should be activated before any other aspects

➢ Order advice
  - The aspect order can be defined by order advice
  - different aspect orders can be defined for different pointcuts

➢ Example
  advice "% util::Queue::%queue(...)"
      : order( "LockingIRQ", "%" && !"LockingIRQ" );

Aspect Instantiation

➢ Aspects are singletons by default
  - aspectof() returns pointer to the one-and-only aspect instance

➢ By overriding aspectof() this can be changed
  - e.g. one instance per client or one instance per thread

aspect MyAspect {
  // ...
  static MyAspect* aspectof() {
    static __declspec(thread) MyAspect* theAspect;
    if( theAspect == 0 )
      theAspect = new MyAspect;
    return theAspect;
  }
};

Example of an user-defined aspectof() implementation for per-thread aspect instantiation by using thread-local storage.
(Visual C++)

Agenda

3.1 Motivation: Separation of Concerns
3.2 Tutorial: AspectC++
  Example Szenario
  First Steps And Language Overview
  Advanced Concepts
  Weaver Transformations
  Further Examples
3.3 Summary and Outlook
3.4 References
Weaver Transformations

Aspect Transformation

```cpp
aspect Transform {
  advice call("% foo()") : before() {
    printf("before foo call\n");
    printf(tjp->signature ());
  }
  template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
      printf (tjp->signature ());
    }
};
```

One global aspect instance is created by default

Advice becomes a member function

Aspects are transformed into ordinary classes
Weaver Transformations

Aspect Transformation

aspect Transform {
  advice call("% foo()") : before() {
    printf("before foo call\n");
  }
  advice execution("% C::%()") : after()
  {
    printf(tjp->signature ());
  }
};

Transform.ah'
“Generic Advice” becomes a template member function

Class Transform {
  static Transform __instance;
  // ...
  void __a0_before () {
    AC::..._a0_before ();
    ::foo();
  }
};

main.cc
int main() {
  struct __call_main_0_0 {
    static inline void invoke (){ AC::..._a0_before ();
      ::foo();
    }
  };
  __call_main_0_0::invoke ();
  return 0;
}

Joinpoint Transformation

main.cc
int main() {
  foo();
  return 0;
}
main.cc'
a local class invokes the advice code for this joinpoint

Joinpoint Transformation

main.cc
int main() {
  foo();
  return 0;
}
main.cc'
the function call is replaced by a call to a wrapper function

Joinpoint Transformation

main.cc
int main() {
  foo();
  return 0;
}
main.cc'

Joinpoint Transformation

main.cc
int main() {
  foo();
  return 0;
}
main.cc'

Joinpoint Transformation

main.cc
int main() {
  foo();
  return 0;
}
main.cc'
a local class invokes the advice code for this joinpoint
Weaver Transformations

**Translation Modes**

- **Whole Program Transformation-Mode**
  - e.g. `ac++ -p src -d gen -e cpp -Iinc -DDEBUG`
  - transforms whole directory trees
  - generates manipulated headers, e.g. for libraries
  - can be chained with other whole program transformation tools

- **Single Translation Unit-Mode**
  - e.g. `ac++ -c a.cc -o a-gen.cc -p .`
  - easier integration into build processes

---

**Agenda**

3.1 Motivation: Separation of Concerns
3.2 Tutorial: AspectC++
   - Example Scenario
   - First Steps And Language Overview
   - Advanced Concepts
   - Weaver Transformations
   - Further Examples
3.3 Summary and Outlook
3.4 References

---

**Further Examples**

**Observer Pattern: Scenario**

```
DigitalClock
Draw()

AnalogClock
Draw()

ClockTimer
GetHour() : int
SetTime (in h : int, in m : int, in s : int)
Tick()
```

```
update on change
```

**Observer Pattern: Implementation**

```
DigitalClock
Draw()

AnalogClock
Draw()

ClockTimer
GetHour() : int
SetTime (in h : int, in m : int, in s : int)
Tick()
```

```
update (in s : ISubject)
IObserver
updateAll()
ISubject
observers
update (in s)
```
Further Examples

Observer Pattern: Problem

The 'Observer Protocol' Concern...
...crosscuts the module structure

Solution: Generic Observer Aspect

aspect ObserverPattern {
  ...  
public:
  struct ISubject {};
  struct IObserver ... ) { ... }
  void remObserver( ISubject* subject, IObserver* observer ) { ... }
};

Interfaces for the subject/observer roles

Further Examples

Solution: Generic Observer Aspect

aspect ObserverPattern {
  ...  
public:
  struct ISubject {};
  struct IObserver ... subject, IObserver* observer ) { ... }
  void remObserver( ISubject* subject, IObserver* observer ) { ... }
};

Further Examples

Solution: Generic Observer Aspect

aspect ObserverPattern {
  ...  
public:
  struct ISubject {};
  struct IObserver ... subject, IObserver* observer ) { ... }
  void remObserver( ISubject* subject, IObserver* observer ) { ... }
};
Further Examples

**Solution: Generic Observer Aspect**

```cpp
aspect ObserverPattern {
  ...   
public:
  struct ISubject {};
  struct IObserver ... pointcut defining all 
state-changing methods.
(Defaults to the execution of any 
non-const method in subjects)
}

void updateObservers( ISubject* subject );
ISubject* subject = ...;
```

**Introduction of the role 
interface as additional 
baseclass into 
subjects / observers**

```cpp
virtual pointcut defining all 
state-changing methods.
(Defaults to the execution of any 
non-const method in subjects)
```

Further Examples

**Solution: Putting Everything Together**

```cpp
aspect ClockObserver : public ObserverPattern {
  // define the participants
  ... s ) {
      Draw();
    }
  };
};
Applying the Generic Observer Aspect to the clock example
```

Further Examples

**Solution: Putting Everything Together**

```cpp
virtual pointcut defining all 
state-changing methods.
(Defaults to the execution of any 
non-const method in subjects)
```

Further Examples

```cpp
virtual pointcut defining all 
state-changing methods.
(Defaults to the execution of any 
non-const method in subjects)
```
**Win32 Errorhandling: Goals**

- Detect failed calls of Win32 API functions
  - by giving after advice for any call to a Win32 function
- Throw a helpful exception in case of a failure
  - describing the exact circumstances and reason of the failure

Problem: Win32 failures are indicated by a “magic” return value
- magic value to compare against depends on the return type of the function
- error reason (GetLastError()) only valid in case of a failure

<table>
<thead>
<tr>
<th>return type</th>
<th>magic value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>FALSE</td>
</tr>
<tr>
<td>ATOM</td>
<td>(ATOM) 0</td>
</tr>
<tr>
<td>HANDLE</td>
<td>INVALID_HANDLE_VALUE or NULL</td>
</tr>
<tr>
<td>HWND</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**Detecting the failure: Generic Advice**

```cpp
advice call(win32API()) : after () {
  if (isError (*tjp->result()))
    // throw an exception
}
```

**These Win32 API functions may fail!**
Further Examples

Describing the failure: Generative Advice

```cpp
template <int I> struct ArgPrinter {
    template <class JP>
    static void work (JP &tjp, ostream &s) {
        ArgPrinter<I–1>::work (tjp, s);
        s << ", " << *tjp.template arg<I-1>();
    }
};
```

```cpp
template <int I> struct ArgPrinter {
    template <class JP>
    static void work (JP &tjp, ostream &s) {
        ArgPrinter<I–1>::work (tjp, s);
        s << ", " << *tjp.template arg<I-1>();
    }
};
```

advice call(win32API ()) : after () {
  // throw an exception
  std::ostringstream s;
  DWORD code = GetLastError();
  s << "WIN32 ERROR " << code << ...
  s << win32::GetErrorText(code) << ... << tjp->signature() << "WITH: " << ...;
  ArgPrinter<JoinPoint::ARGS>::work (*tjp, s);
  throw win32::Exception (s.str());
}
```

```cpp
advice call(win32API ()) : after () {
  // throw an exception
  std::ostringstream s;
  DWORD code = GetLastError();
  s << "WIN32 ERROR " << code << ...
  s << win32::GetErrorText(code) << ... << tjp->signature() << "WITH: " << ...;
  ArgPrinter<JoinPoint::ARGS>::work (*tjp, s);
  throw win32::Exception (s.str());
}
```

Reporting the Error

```cpp
LRESULT WINAPI WndProc( HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam ) {
    HDC dc = NULL; PAINTSTRUCT ps = {0};
    switch (nMsg) {
        case WM_PAINT:
            dc = BeginPaint( hWnd, &ps );
            ...
            EndPaint(hWnd, &ps);
            break;
        ...
    }
    return 0;
}
```

```cpp
LRESULT WINAPI WndProc( HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam ) {
    HDC dc = NULL; PAINTSTRUCT ps = {0};
    switch (nMsg) {
        case WM_PAINT:
            dc = BeginPaint( hWnd, &ps );
            ...
            EndPaint(hWnd, &ps);
            break;
        ...
    }
    return 0;
}
```

Agenda

3.1 Motivation: Separation of Concerns
3.2 Tutorial: AspectC++
3.3 Summary and Outlook
3.4 Références

Aspect-Oriented Programming: Summary

- AOP aims at a better separation of crosscutting concerns
  - Avoidance of code tangling
    - implementation of optional features
  - Avoidance of code scattering
    - implementation of nonfunctional features
- Basic idea: separation of **what** from **where**
  - **Join Points**  ➔ **where**
    - positions in the static structure or dynamic control flow (event)
    - given declaratively by pointcut expressions
  - **Advice**  ➔ **what**
    - additional elements (members, ...) to introduce at join points
    - additional behavior (code) to superimpose at join points
- AspectC++ brings AOP concepts to the C++ world
  - Static source-to-source transformation approach
Aspect-Oriented Programming: Summary

- AOP aims at a better separation of crosscutting concerns
  - Avoidance of code tangling
  - Implementation of optional features
  - Avoidance of code scattering
  - Implementation of nonfunctional features
- Basic idea: separation of what from where
  - Join Points: where
    - positions in the static structure or dynamic control flow (event)
    - given declaratively by pointcut expressions
  - Advice: what
    - additional elements (members, ...) to introduce at join points
    - additional behavior (code) to superimpose at join points
- AspectC++ brings AOP concepts to the C++ world
  - Static source-to-source transformation approach

Referenzen


**AspectC++ Quick Reference**

**Concepts**

aspect
Aspects in AspectC++ implement in a modular way cross-cutting concerns and are an extension to the class concept of C++. Additionally to attributes and methods, aspects may also contain advice declarations.

advice
An advice declaration is used either to specify code that should run when the join points specified by a pointcut expression are reached or to introduce a new method, attribute, or type to all join points specified by a pointcut expression.

slice
A slice is a fragment of a C++ element like a class. It may be used by introduction advice to implemented static extensions of the program.

**Aspects**

Writing aspects works very similar to writing C++ class definitions. Aspects may define ordinary class members as well as advice.

aspect A {...};
defines the aspect A
aspect A : public B {...};
A inherits from class or aspect B

---

### Advice Declarations

advice pointcut : before(...) {...}
the advice code is executed before the join points in the pointcut
advice pointcut : after(...) {...}
the advice code is executed after the join points in the pointcut
advice pointcut : around(...) {...}
the advice code is executed in place of the join points in the pointcut
advice pointcut : order(high, ...low);
high and low are pointcuts, which describe sets of aspects. Aspects on the left side of the argument list always have a higher precedence than aspects on the right hand side at the join points, where the order declaration is applied.

advice pointcut : slice class : public Base {...}
introduces a new base class Base and members into the target classes matched by pointcut.

advice pointcut : slice ASlice;
introduces the slice ASlice into the target classes matched by pointcut.

### Pointcut Expressions

**Type Matching**

"int"
matches the C++ built-in scalar type int
"% *"
matches any pointer type

**Namespace and Class Matching**

"Chain"
matches the class, struct or union Chain
"Memory%"
matches any class, struct or union whose name starts with “Memory”

**Function Matching**

"void reset()"
matches the function reset having no parameters and returning void
"% printf(...)"
matches the function printf having any number of parameters and returning any type
"% ...::%(...)"
matches any function, operator function, or type conversion function (in any class or namespace)

---

### Predefined Pointcut Functions

**Functions**

**call**(pointcut)
N→C‡‡
provides all join points where a named entity in the pointcut is called.

**execution**(pointcut)
N→C‡
provides all join points referring to the implementation of a named entity in the pointcut.

**construction**(pointcut)
N→C_{cons}
all join points where an instance of the given class(es) is constructed.

**destruction**(pointcut)
N→C_{dest}
all join points where an instance of the given class(es) is destructed.

**pointcut** may contain function names or class names. A class name is equivalent to the names of all functions defined within its scope combined with the :: operator (see below).

**Control Flow**

cflow(pointcut)
C→C captures join points occurring in the dynamic execution context of join points in the pointcut. The argument pointcut is forbidden to contain context variables or join points with runtime conditions (currently cflow, that, or target).

**Types**

base(pointcut)
N→N_{r}
returns all base classes resp. redefined functions of classes in the pointcut

derived(pointcut)
N→N_{r}
returns all classes in the pointcut and all classes derived from them resp. all redefined functions of derived classes
**Scope**

within(pointcut) \( N \rightarrow C \)

filters all join points that are within the functions or classes in the pointcut

**Context**

that(type pattern) \( N \rightarrow C \)

returns all join points where the current C++ \texttt{this} pointer refers to an object which is an instance of a type that is compatible to the type described by the type pattern

target(type pattern) \( N \rightarrow C \)

returns all join points where the target object of a call is an instance of a type that is compatible to the type described by the type pattern

result(type pattern) \( N \rightarrow C \)

returns all join points where the result object of a call/execution is an instance of a type described by the type pattern

args(type pattern, ...) \((N...) \rightarrow C\)

a list of type patterns is used to provide all joinpoints with matching argument signatures

Instead of the type pattern it is possible here to pass the name of a context variable to which the context information is bound. In this case the type of the variable is used for the type matching.

**Algebraic Operators**

pointcut \&\& pointcut

intersection of the join points in the pointcuts

pointcut \|\| pointcut

union of the join points in the pointcuts

! pointcut

exclusion of the join points in the pointcuts

**JoinPoint-API**

The JoinPoint-API is provided within every advice code body by the built-in object \texttt{tjp} of class \texttt{JoinPoint}.

**Compile-time Types and Constants**

That

object type (object initiating a call)

```
[\text{type}]
```

Target

target object type (target object of a call)

```
[\text{type}]
```

Result

type of the object, which is used to store the result of the affected function

```
[\text{type}]
```

Res::Type, Res::ReferredType

result type of the affected function

```
[\text{type}]
```

**Runtime Functions and State**

**Runtime Type Information**

```
\begin{align*}
\text{static AC::Type resulttype}() \\
\text{static AC::Type argtype}(\text{int } i) \\
\end{align*}
```

return a C++ ABI V3 conforming string representation of the result type / argument type of the affected function

**Example**

A reusable tracing aspect.

```cpp
class Trace {
    void proceed();
    ...}.
```

In a derived aspect the pointcut \texttt{functions} may be redefined to apply the aspect to the desired set of functions.

```cpp
aspect TraceMain : public Trace {
    pointcut functions() = "% main(...)";
};
```

This is a reference sheet corresponding to AspectC++ 1.0. Version 1.12, April 18, 2011.

(c) Copyright 2011 pure-systems GmbH, Olaf Spinczyk and Daniel Lohmann. All rights reserved.