The Aspect-Oriented Design of the PUMA C/C++ Parser Framework

Matthias Urban
pure systems GmbH
Magdeburg

Daniel Lohmann
Friedrich-Alexander University
Erlangen-Nuremberg

Olaf Spinczyk
Technical University
Dortmund

9th International Conference on Aspect-Oriented Software Development
Industry Track – March 16, 2010
What is **PUMA**

**PUMA**

A generic framework for applications that have to

parse, analyze, and optionally transform

various flavors of C and C++ source code.
What is PUMA

PUMA

A generic framework for applications that have to parse, analyze, and optionally transform various flavors of C and C++ source code.

- freely available under the GPL
  - https://svn.aspectc.org/repos/Puma/trunk
  - 83,000 lines of code
- developed and maintained by pure::systems GmbH, Magdeburg
  - internally used for the development of client-specific solutions
  - commercial licenses and support available
- used by – and implemented in – AspectC++
PUMA Application Examples

The **AspectC++ weaver ac++**

```
aspect Cool {
   ...;
   int main() {
      ...;
   }
}
```

```
... int main() {
   _Cool_invoke_a0();
   ...;
}
```

---

dl@cs.fau.de  The PUMA C/C++ Parser Framework (AOSD '10)  Introduction
Puma Application Examples

The AspectC++ weaver ac++

A mutation testing tool for SystemC
Concerns of a C/C++ Parser

Primary job of a parser (greatly simplified)
- read tokens from input stream (keywords, identifier, operator symbols)
- invoke matching grammar rules

Additional concerns (there are many!)
- syntax tree construction
- tentative parsing
- error handling
- connection to the semantic analysis
- lookahead optimizations
Concerns of a C/C++ Parser

- Primary job of a parser (greatly simplified)
  - read tokens from input stream
    (keywords, identifier, operator symbols)
  - invoke matching grammar rules

- Additional concerns (there are many!)
  - syntax tree construction
  - tentative parsing
  - error handling
  - connection to the semantic analysis
  - lookahead optimizations

State of the Art (for example, gcc/g++)

All these concerns are tangled and scattered in the implementation!
The Challenge

512 pages
The Challenge

512 pages

816 pages of spec!
The Challenge

512 pages

816 pages of spec!

700 pages (without .NET!)
The Challenge

512 pages

816 pages of spec!

647 pages

700 pages (without .NET!)
A Family of C/C++ Parsers and Manipulators

Input Languages
- C++
- C
- Dialects
  - AspectC++
  - MS Visual C++
  - GNU gcc/g++

Analyses
- CPP Parsing & Sem. Analysis
  - Full Sem. Analysis
- Transformation
  - AST Matching
A Family of C/C++ Parsers and Manipulators

Input Languages
- C++
- C
- Dialects
  - AspectC++
  - MS Visual C++
  - GNU gcc/g++

Analyses
- CPP Parsing & Sem. Analysis
- Full Sem. Analysis

Transformation
- AST Matching
A Family of C/C++ Parsers and Manipulators

Puma

Input Languages

C++
C
Dialects

Analyses

CPP Parsing & Sem. Analysis
Full Sem. Analysis
AST Matching

Transformation

AspectC++
MS Visual C++
GNU gcc/g++
Full Sem. Analysis
A Family of C/C++ Parsers and Manipulators

Input Languages
- C++
- C
- Dialects
- AspectC++
- MS Visual C++
- GNU gcc/g++

Analyses
- CPP Parsing & Sem. Analysis
- Full Sem. Analysis
- AST Matching

Transformation
A Family of C/C++ Parsers and Manipulators

Input Languages
- C++
- C

Dialects
- AspectC++
- MS Visual C++
- GNU gcc/g++

Analyses
- CPP Parsing & Sem. Analysis
- Full Sem. Analysis

Transformation
- AST Matching

Puma
A Family of C/C++ Parsers and Manipulators

Input Languages
- C++
- C

Analyses
- CPP Parsing & Sem. Analysis
- Full Sem. Analysis

Transformation
- AST Matching

Dialects
- AspectC++
- MS Visual C++
- GNU gcc/g++
A Family of C/C++ Parsers and Manipulators

- **Input Languages**
  - C++
  - C
  - Dialects
    - AspectC++
    - MS Visual C++
    - GNU gcc/g++

- **Analyses**
  - CPP Parsing & Sem. Analysis
  - Full Sem. Analysis

- **Transformation**
  - AST Matching

---

The Puma C/C++ Parser Framework (AOSD '10) Introduction 6
A Family of C/C++ Parsers and Manipulators

Input Languages
- C++
- C
- Dialects
  - AspectC++
  - MS Visual C++
  - GNU gcc/g++

Analyses
- CPP Parsing & Sem. Analysis
- Full Sem. Analysis
- AST Matching

Transformation
A Family of C/C++ Parsers and Manipulators

- **Input Languages**
  - C++
  - C
  - Dialects
    - AspectC++
    - MS Visual C++
    - GNU gcc/g++

- **Analyses**
  - CPP Parsing & Sem. Analysis
  - Full Sem. Analysis

- **Transformation**
  - AST Matching

---

dl@cs.fau.de  The PUMA C/C++ Parser Framework (AOSD '10)  Introduction  6
A Family of C/C++ Parsers and Manipulators

Goal: Configurability and Extensibility
- Separation of concerns crucial for success!
- Aspect-oriented design and implementation
Focus of this Talk: PUMA Parsers

Input Languages
- C++
- C
- Dialects
- AspectC++
- MS Visual C++
- GNU gcc/g++

Analyses
- CPP Parsing & Sem. Analysis
- Full Sem. Analysis

Transformation
- AST Matching

Goal: Configurability and Extensibility
- Separation of concerns crucial for success!
- Aspect-oriented design and implementation

dl@cs.fau.de
Agenda

1. Introduction
2. Design Methodology
3. Separation of Concerns in PUMA
4. Achievements
5. Wrap up
Provider–Consumer Relationship **without** AOP

- Event provider has to **know** event consumer
- Control flows specified **in** the direction of knowledge
Obliviousness and Quantification demystified

Provider–Consumer Relationship with AOP

- Event consumer has to **know** event provider
- Advice specifies control flows **against** the direction of knowledge
  ➡️ the mechanism behind “obliviousness”
Obliviousness and Quantification demystified

Provider–Consumer Relationship with AOP

- Advice specifies control flows against the direction of knowledge
- Control flow specification is inherently loose
  ➔ the mechanism behind “quantification”
Methodology: Aspect-Aware System Design

Basic Idea: Separation of Concerns in the Implementation

- **one feature** per implementation unit
- strict **decoupling** of policies and mechanisms
- use aspects as **primary** composition technique
Methodology: Aspect-Aware System Design [USENIX 09]

Basic Idea: Separation of Concerns in the Implementation

- **one feature** per implementation unit
- **strict decoupling** of policies and mechanisms
- use aspects as **primary** composition technique

Design Principles          Development Idioms

1. **loose coupling**  
   - by advice-based binding
2. **visible transitions**  
   - by explicit join points
3. **minimal extensions**  
   - by extension slices

dl@cs.fau.de  The PUMA C/C++ Parser Framework (AOSD ’10)  Design Methodology
Basic Idea: Separation of Concerns in the Implementation

- one feature per implementation unit
- strict decoupling of policies and mechanisms
- use aspects as primary composition technique

Design Principles \[\mapsto\] Development Idioms

1. loose coupling \[\mapsto\] by advice-based binding
2. visible transitions \[\mapsto\] by explicit join points
3. minimal extensions \[\mapsto\] by extension slices

\[\mapsto\] we partly give up the obliviousness idea!
A Minimal Example

(Also: AspectC++ in 2 Minutes)

class PreprocessorParser implements the ISO standard aspect GNUMacros extends it by gcc/g++'s predefined macros GnuMacros is a minimal extension brought in as an extension slice into class PreprocessorParser integrated by advice-based binding to configure()

dl@cs.fau.de The PUMA C/C++ Parser Framework (AOSD '10) Design Methodology 10
A Minimal Example

class PreprocessorParser implements the ISO standard
aspect GNUMacros extends it by gcc/g++’s predefined macros

- GnuMacros is a minimal extension
- brought in as an extension slice into class PreprocessorParser
- integrated by advice-based binding to configure()
Methodology: Roles of Aspects and Classes

What to model as a class and what as an aspect?

- `<thing>` is modelled as a class if – and only if – it is a distinguishable, instantiable concept of PUMA:
  - a system component, instantiated internally on behalf of PUMA
  - a system abstraction, instantiated as objects on behalf of the user
  - both are sparse \(\mapsto\) provide a minimal implementation only

- otherwise `<thing>` is an aspect!

- we came up with three idiomatic aspect roles
  - extension aspects
  - policy aspects
  - upcall aspects
Agenda

1. Introduction

2. Design Methodology

3. Separation of Concerns in Puma

4. Achievements

5. Wrap up
Aspect-Aware System Abstractions

Syntax

ISO C

literal:
  identifier

primary-expression:
  literal
  ( expression )

expression:
  primary-expression
  expression + primary-expression

ISO C++

literal:
  identifier
  true
  false

primary-expression: same as in C
expression: same as in C
struct CSyntax : public Syntax {

    struct Literal {
        static bool check (CSyntax &s) { return s.literal(); }
        static bool parse (CSyntax &s) { return s.token(ID); }
    };

    virtual bool literal() { return Literal::parse(*this); }

    struct Primary {
        static bool check (CSyntax &s) { return s.primary(); }
        static bool parse (CSyntax &s) {
            return Literal::check(s) ||
                (s.token(')') && Expr::check(s) && s.token(')'));
        }
    };

    virtual bool primary() { return Primary::parse(*this); }
    ...
Aspect-Aware System Abstractions

```cpp
struct CSyntax : public Syntax {

  struct Literal {
    static bool check (CSyntax &s) { return s.literal(); }
    static bool parse (CSyntax &s) { return s.token(ID); }
  };

  virtual bool literal() { return Literal::parse(*this); }

  struct Primary {
    static bool check (CSyntax &s) { return s.primary(); }
    static bool parse (CSyntax &s) {
      return Literal::check(s) || (s.token('(') && Expr::check(s) && s.token(')'));
    }
  };

  virtual bool primary() { return Primary::parse(*this); }
};
```

ISO C

- literal: identifier
- primary-expression: literal ( expression )
- expression: primary-expression expression + primary-expression

Grammar rule

- a virtual function and an inner class
- virtual function ↦ delegates implementation to inner class
- inner class ↦ unambiguous scope for minimal extensions
- class members ↦ unambiguous join points for visible transitions
struct CSyntax : public Syntax {

struct Literal {
    static bool check (CSyntax &s) { return s.literal(); }
    static bool parse (CSyntax &s) { return s.token(ID); }
};

virtual bool literal() { return Literal::parse(*this); }

struct Primary {
    static bool check (CSyntax &s) { return s.primary(); }
    static bool parse (CSyntax &s) {
        return Literal::check(s) ||
            (s.token('(') && Expr::check(s) && s.token(')'));
    }

    virtual bool primary() { return Primary::parse(*this); }

    ...
};

Grammar rule $\mapsto$ a virtual function and an **inner class**

- virtual function $\mapsto$ delegates implementation to inner class
- inner class $\mapsto$ unambiguous scope for **minimal extensions**
- class members $\mapsto$ unambiguous join points for **visible transitions**
SyntaxState implements a mandatory, but crosscutting policy
- currently matches 104/118 grammar rules in the C/C++ syntax
- implementation as aspects keeps backtracking policy configurable

inherently scales with extension aspects that add more rules
- VisualC++, GNU, AspectC++, OpenMP, MPI, C++1x, ...
CBuilder extends grammar rules for syntax tree construction
- each rule gets a build() method to construct corresponding element
**Extension/Upcall Aspects: Syntax Tree Construction**

- **Builder** extends grammar rules for syntax tree construction
  - each rule gets a `build()` method to construct corresponding element

- **Builder** binds this extension by **upcalls**
  - each rule invocation by the parser now also triggers `build()`
Dialect aspects extend both, CSyntax and CCSyntax
- facilitates implicit reuse of C extensions in the C++ parser
**Extension Aspects: Language Extensions**

- **Dialect aspects** extend both, **CSyntax** and **CCSyntax**
  - facilitates implicit reuse of C extensions in the C++ parser

- **Example for loose coupling**
  - In C-only projects, **CCSyntax** is just not present
    - → C++ extensions are silently skipped
  - inherent property of **advice-based binding**
1 Introduction

2 Design Methodology

3 Separation of Concerns in PUMA

4 Achievements

5 Wrap up
Separation of Concerns \(\leadsto\) Configurability

Achieved: Complete “Plug & Play” of Features

Loose coupling of feature implements distinct sets of implementation units integrate themselves by advice-based binding.
Achieved: Complete “Plug & Play” of Features

**Loose coupling** of feature *implements*

- distinct sets of implementation units
- integrate themselves by *advice-based binding*
Separation of Concerns $\leadsto$ Maintainability

LoC taken by the **C parser** and **C++ parser** implementations

**Puma**
- CPParser.cc: 1,786
- CCParser.cc: 2,802

**GNU**
- c-parser.c: 8,676
- cpp-parser.cc: 22,964
### Separation of Concerns ⟷ Maintainability

LoC taken by the **C parser** and **C++ parser** implementations

<table>
<thead>
<tr>
<th></th>
<th>CParser.cc</th>
<th>CCPParser.cc</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Puma</strong></td>
<td>1,786</td>
<td>2,802</td>
</tr>
<tr>
<td>GNU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c-parser.c</td>
<td></td>
<td>8,676</td>
</tr>
<tr>
<td>cpp-parser.cc</td>
<td></td>
<td>22,964</td>
</tr>
</tbody>
</table>

**Achieved: Cost Effectiveness**

Even though **PUMA** is a complex piece of software, it remains maintainable by a **single, part-time engineer**.
Separation of Concerns $\overset{\sim}{\rightarrow}$ Extensibility

An unanticipated Extension: C++ 1x static assertions

```cpp
static_assert ( constant-expr, error-message ) ;
```
Separation of Concerns ⇝ Extensibility

An unanticipated Extension: C++ 1x static assertions

```c
static_assert ( constant-expr, error-message ) ;
```

We could translate the impact description ("proposed wording") of this new C++ feature almost literally into aspect code

- policy aspect for binding the code and enabling/disabling it by a new command line options [LoC] 112
- syntax tree class for static assertions 102
- syntax rule (extension slice) 77
- syntax tree creation (extension slice) 73
- function for semantic analysis (extension slice) 100
Separation of Concerns $\leadsto$ Extensibility

An unanticipated Extension: \texttt{C++ 1x static assertions}

\begin{verbatim}
static assert ( constant-expr, error-message ) ;
\end{verbatim}

“A compiler writer could certainly implement this feature, as specified, in two or three days…”
Separation of Concerns \(\rightsquigarrow\) Extensibility

An unanticipated Extension: C++ 1x static assertions

\[
\text{static assert (constant-expr, error-message)};
\]

“A compiler writer could certainly implement this feature, as specified, in two or three days…”

It took us just one day!

- including documentation and testing
- including extra work for the very first C++ 1x feature
Agenda

1. Introduction
2. Design Methodology
3. Separation of Concerns in PUMA
4. Achievements
5. Wrap up
A Parser for “real-world” C/C++ code is a complex thing
- Many concerns to deal with
- C++ is one of the most challenging languages at all
- Various language dialects and extensions

PUMA copes well with this variability and complexity
- Relatively small code base, separation of concerns
- Configurability
- Extensibility

Achieved by AOP and an aspect-aware design
- Sparse classes, explicit join points
- Employed aspects as the fundamental extension and binding mechanism
- “Plug&Play” configuration and extension
Future Work

- Advertise it more! :-)
  - **PUMA** is the *best* and most *feature-complete* open-source C++ analysis and transformation framework we are aware of!
    (BTW: commercial support is available as well...)  
  - But it is still a bit hidden (published as part of AspectC++)

- Incorporate C++ 1x features
  - Incrementally, start with the most probable ones
  - A lot of work, but we are now *optimistic* that this is feasible!

Get **PUMA** at:

https://svn.aspectc.org/repos/Puma/trunk