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

Matthias Urban
pure-systems GmbH
Magdeburg, Germany
matthias.urban@pure-systems.com

Daniel Lohmann
FAU Erlangen-Nürnberg
Computer Science 4
dl@cs.fau.de

Olaf Spinczyk
TU Dortmund
Computer Science 12, ESS
olaf.spinczyk@udo.edu

ABSTRACT

PUMA is a framework for the development of applications that analyze and, optionally, transform C or C++ source code. It supports ISO C and C++ as well as many language extensions of the GNU Compiler Collection and Microsoft Visual C++. Aspects played an important role during the design and implementation of the framework. It is written in the AspectC++ language. By employing AOSD concepts, we gained a clean separation of concerns and, thereby, very good configurability and extensibility. All these -ilities are of vital importance for our project, because the available manpower for maintenance tasks is limited. This paper briefly describes the design principles behind PUMA.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Parsing; D.2.2 [**Design Tools and Techniques**]; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Design

Keywords

Aspect-Oriented Programming (AOP), Aspect-Oriented Design, AspectC++, PUMA

1. INTRODUCTION

PUMA has been developed by pure-systems GmbH, a company located in Magdeburg, Germany, where it is applied in the development of client-specific tools for the analysis/transformation of C/C++ source code. For instance, one recent PUMA-based project was a mutation testing tool for SystemC [13] code. With PUMA the development of this tool could be simplified significantly. Even though PUMA is being used in commercial projects, it is also available under the GPL[1]. This is remarkable, because as far as we know there are no other open-source parsers for C++ available that are as feature complete as PUMA.

A well-known noncommercial application of PUMA is the AspectC++ weaver ac++. AspectC++ [12] is an aspect-oriented extension of C++ and, at the same time, the implementation language of PUMA.

Figure 1 gives an overview of the features that PUMA provides for its applications. As several features are configurable, we regard PUMA not as a single framework but as a product line [3] of frameworks.

With about 83,000 lines of code, PUMA is a complex piece of software. In the following, we concentrate on the aspect-oriented *design* principles applied in the construction of PUMA *parsers*, which is – with respect to separation of concerns and extensibility – the most interesting and challenging part of PUMA.

1.1 Crosscutting Concerns of C/C++ Parsers

A parser has to accomplish various tasks at the same time. Primarily it has to read tokens (such as identifiers, keywords, or operator symbols) from an input stream and match them against a set of grammar rules. The rules can either be implemented by (generated) tables or by (hand written) functions. For performance and complexity reasons, the latter is the common approach in the C/C++ domain. The GNU gcc/g++ parser, for instance, is a huge piece of hand-written C code; all grammar rules are expressed by C functions that call each other. However, besides the grammar implementation, there are many other concerns:

- Language extensions: Objective-C and OpenMP (implemented by conditional code)
- Construction of syntax tree nodes
- "Tentative Parsing" (speculative parsing and backtracking)
- Connection to the semantic analysis
- Error handling
- Specific look-ahead optimizations

In gcc/g++, all these concerns are an integral part of the C/C++ grammar. The code is tangled, which probably makes it difficult to maintain and especially extend the parser. Furthermore, there is no re-use – even though parts of the C++ grammar are very similar to the C grammar.

[*]This work was partly supported by the German Research Council (DFG) under grant no. SCHR 603/4, SP 968/2-1, and SP 968/4-1.

[1] PUMA is available as a part of the AspectC++ weaver ac++. It's latest version can be anonymously downloaded via subversion from the URL `https://svn.aspectc.org/repos/Puma/trunk`.

AOSD '10 March 15-19, 2010, Rennes and St. Malo, France.

LOCs	GNU gcc/g++		PUMA	
C	c-parser.c	**8676**	CSyntax.cc	**1786**
C++	cpp-parser.cc	**22964**	CCSyntax.cc	**2802**

Table 1: Comparison between the C/C++ syntax rule implementations in GNU gcc/g++ and Puma

The aspect-oriented design approach taken in the development of PUMA leads to much better maintainability and extensibility, as indicated in Table 1 by the comparison of the lines of code taken by the respective C and C++ grammar implementations.

1.2 Primary Goal: Extensibility

A primary design goal behind PUMA is its extensibility. We have to deal with various language extensions and vendor-specific dialects, such as AspectC++, GNU and MS Visual C++, as well as with a number of different C and C++ standards, such as C99, C++1x, and so on. Additionally, PUMA is intended to support client-specific language extension. Therefore, the design must focus on extensibility as a key feature.

2. DESIGN METHODOLOGY

During the last eight years we have developed aspect-oriented software product lines with AspectC++ for various purposes, including operating systems [9] and resource-constrained embedded systems [10]. Over the years the insights from these projects evolved into the design methodology that has also been used for PUMA. In the following, we briefly present its core ideas (for further details see [8]).

2.1 Principles

The basic idea behind our design approach is the strict separation of concerns in the *implementation*. Each implementation unit provides exactly one feature; its mere presence or absence in the configured source tree decides on the inclusion of the particular feature into the resulting system variant.

Technically, this comes down to a strict decoupling of policies and mechanisms by using aspects as the primary composition technique: Mechanisms are glued together and extended by aspects; they support aspects by ensuring that all relevant internal control-flow transitions are available as unambiguous and statically evaluable join points.

We learned from this that the exposure of all relevant gluing and extension points as statically evaluable and unambiguous join points has to be understood as a primary design goal from the very beginning. The key premise for such *aspect awareness* is a component structure that makes it possible to influence the composition and shape of components as well as all run-time control flows that run through them by aspects. This led to the following design principles for PUMA:

The principle of loose coupling. Make sure that aspects can hook into all facets of the static and dynamic *integration* of system components.

The principle of visible transitions. Make sure that aspects can hook into all control flows that run through the system. All control-flow transitions into, out of, and within the system should be influenceable by aspects. For this they have to be represented on the join-point level as statically evaluable, unambiguous join points.

The principle of minimal extensions. Make sure that aspects can extend all features provided by the system on a fine granularity. System components and system abstractions should be fine-grained, sparse, and extensible by aspects.

Aspect awareness, as described by these principles, means that we moderate the AOP ideal of obliviousness, which is generally considered by the AOP community as a defining characteristic of AOP. PUMA's system components and abstractions are *not* totally oblivious to aspects – they are supposed to provide explicit support for aspects and even depend on them for their integration.

2.2 Role and Types of Classes and Aspects

The relationship between aspects and classes is asymmetrical in most AOP languages including AspectC++: Aspects augment classes, but not vice versa. This gives rise to the question which features are best to be implemented as classes and which as aspects and how both should be applied to meet the above design principles.

The general rule we came up with in the development of PUMA and other systems is to provide some feature as a class if – and only if – it represents a *distinguishable instantiable concept* of the system. Provided as classes are:

1. **System Components**, which are instantiated on behalf of PUMA and manage its run-time state (such as the `UnitManager`, which maintains a list of opened and scanned source code files in PUMA).

2. **System Abstractions**, which are instantiated on behalf of the application and represent a system object (such as the C or C++ parser).

However, the classes for system components and system abstractions are sparse and to be further "filled" by *extension slices*. The main purpose of these classes is to provide a distinct scope with unambiguous join points for the aspects (that is, *visible transitions*).

All other features are implemented as aspects. During the development of PUMA we came up with three idiomatic roles of aspects:

1. **Extension Aspects** add additional features to a system abstraction or component (*minimal extensions*), such as extending the preprocessor by GNU compiler-specific predefined macros.

2. **Policy Aspects** "glue" otherwise unrelated system abstractions or components together to implement some policy (*loose coupling*). For instance, handling syntax errors during the parse process can very well be handled by glueing the component that generates error messages with the appropriate rules of the grammar by a policy aspect.

3. **Upcall Aspects** bind behavior defined by higher layers to events produced in lower layers of the system, such as intercepting the execution of the preprocessor `configure` function by the higher-level GNU extension.

One aspect in the implementation can fulfill multiple roles. The effect of *extension aspects* typically becomes visible in the API of the affected system component or abstraction. *Policy aspects*, in contrast, lead to a different system behavior. We will see examples for extension and policy aspects in the following section. *Upcall aspects* do not contribute directly to a design principle, but have a more technical purpose: they exploit advice-based binding and the fact that AspectC++ inlines advice code at the respective join point for flexible, yet very efficient, upcalls.

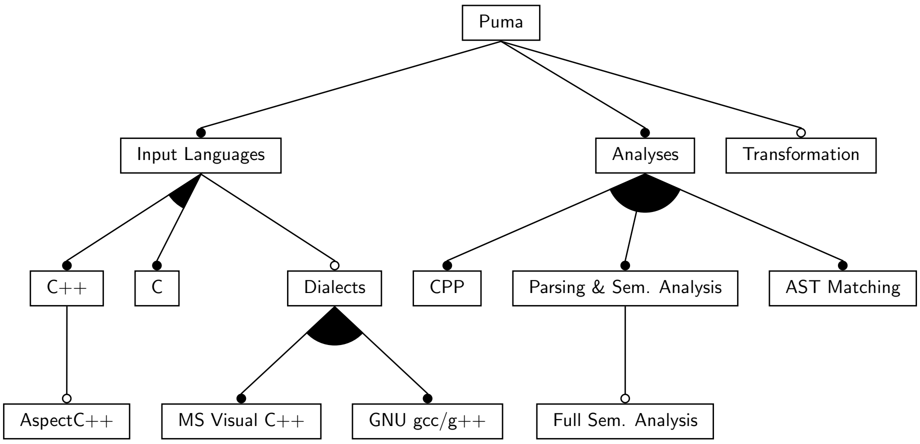


Figure 1: PUMA's feature model: Supported input languages are C and C++; AspectC++ is an optional extension. The Microsoft Visual C++ and GNU gcc/g++ dialects are optional, too. Source code analyses consist of preprocessing (CPP) and parsing. The parser already performs most of the semantic analysis. Additional semantic analyses are provided as an optional sub-feature. AST Matching is a mechanism for searching syntactical patterns. Support for source code transformation is optional as well.

3. SEPARATION OF CONCERNS IN PUMA

This section briefly describes the design of the PUMA C and C++ parsers, which is following the design principles introduced in the previous section. The design process led to clean separation of all relevant parser concerns identified in Section 1.1. We present the aspect-oriented implementations of the *backtracking* and the *syntax tree construction* concerns as examples in order to illustrate the three roles of aspects.

3.1 C and C++ Syntax

The syntax of C and C++ is precisely defined in the respective ISO standards [5, 6]. As most other C/C++ parsers, the PUMA parser is hand written and not generated. Because many of the syntax rules of C are also used in C++ with only minor extensions, we decided to avoid redundancy and, thus, expressed the C and C++ syntax as classes with an inheritance relation (see Figure 2).

The base class `Syntax` provides the interface to the scanner via the `token()` function. All implementations of grammar rules are `virtual` functions in order to allow a derived grammar implementation to override the rule. This is done for `literal()` in our simplified example grammar. All functions return a `bool` value that is `true` if the input token stream matches the implemented rule and `false` otherwise.

We regard this system structure as *aspect-aware*, because it obeys the aforementioned design principles. For instance, grouping operations that are related to the same grammar rule in a common inner class, e.g. `CSyntax::Literal`, is important for *loose coupling* and *visible transitions*. All relevant events, such as the execution of a grammar rule, the invocation of another rule, a base-class rule invocation, or the dynamic dispatch, are accessible for aspects in an *unambiguous* manner. The disadvantage of this implementation is that some more source code has to be written. However, this is *the only* disadvantage. It is purely mechanic work and was done quickly. Thanks to function inlining by the C++ compiler, the additional structuring does not cause any overhead on the machine code level.

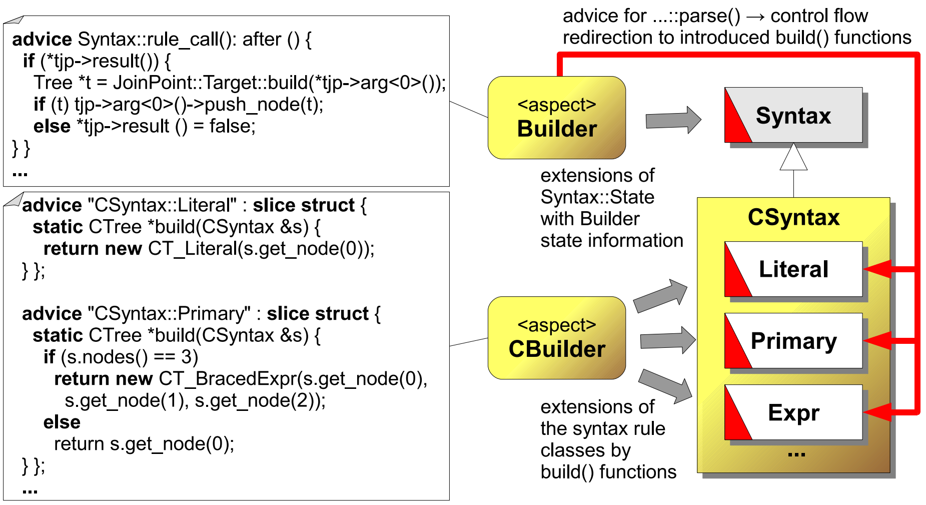
Based on the homogeneous structure of the implementation and the unambiguous join points we can define a number of pointcuts as members of the `Syntax` class. This is shown in Figure 3. The pointcuts can be regarded as an explicit representation of an inter-

```
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); }

  // ... struct Expr not shown here
};

class CCSyntax : public CSyntax {
  struct Literal : public CSyntax::Literal {
    static bool parse (CCSyntax &s) {
      return CSyntax::Literal::parse(s) || s.token(TRUE) ||
        s.token(FALSE);
    }
  };
  virtual bool literal() { return Literal::parse(*this); }
};
```

Figure 2: Aspect-aware implementation of the grammar rules.

```
class Syntax {
  // ...
  pointcut parse_fct()  = "bool %::%::parse(%)";
  pointcut check_fct()  = "bool %::%::check(%)";
  pointcut in_syntax()  = within(derived("Syntax"));
  // rule_exec: execution of a parse function
  pointcut rule_exec()  = execution(parse_fct()) && in_syntax();
  // rule_call: call of a parse function after a dynamic dispatch
  pointcut rule_call()  = call(parse_fct()) && in_syntax() &&
                          !within("%::...::%");
  // rule_check: a rule checks a sub-rule (before dynamic dispatch)
  pointcut rule_check() = execution(check_fct()) && in_syntax();
};
```

Figure 3: An explicit interface for aspects



```
advice Syntax::rule_call(): after () {
  if (*tjp->result()) {
    Tree *t = JoinPoint::Target::build(*tjp->arg<0>());
    if (t) tjp->arg<0>()->push_node(t);
    else *tjp->result () = false;
} }
...
```

```
advice "CSyntax::Literal" : slice struct {
  static CTree *build(CSyntax &s) {
    return new CT_Literal(s.get_node(0));
} };

advice "CSyntax::Primary" : slice struct {
  static CTree *build(CSyntax &s) {
    if (s.nodes() == 3)
      return new CT_BracedExpr(s.get_node(0),
        s.get_node(1), s.get_node(2));
    else
      return s.get_node(0);
} };
...
```

Figure 4: Aspects for syntax tree construction

```
aspect SyntaxState {
  // intercept all calls of rules (after dynamic dispatch)
  advice Syntax::rule_call() : around () {
    Syntax &s = *tjp->arg<0>(); // get 0th argument (CSyntax obj.)
    Syntax::State state;         // local variable to store the state
    s.get_state(state);          // save current parser state
    tjp->proceed();              // perform the intercepted call
    if (!*tjp->result())         // check whether result is false
      s.set_state(state);         // restore the state
  }
};
```

Figure 5: Backtracking support implemented as an aspect

face for aspects, which need to be activated when events occur that are related to the execution of the syntax rules.

3.2 Backtracking and Scanner State

The simple parser described so far only works correctly if the `token()` function does not consume the current token in cases where it does *not* match its argument and if the grammar is LL(1), which means that one token look-ahead is sufficient to decide which production of a grammar rule is the right one. For C and C++ the latter precondition does not hold. Therefore, the parser has to deal with backtracking, which means that the state of the scanner has to be saved when a rule is entered and restored afterwards if the result is `false`. Otherwise an alternative rule would not be tried with the same input tokens as the first rule, which failed.

Because of the aspect-aware system structure (Figure 2) and the aspect interface (Figure 3), this can be expressed easily as an aspect in AspectC++ (see Figure 5). The aspect `SyntaxState` is a *policy aspect*, because it connects the syntax rules with the functions to save and restore the parser state and decides under which circumstances this should happen. For instance, a more sophisticated implementation of this policy could avoid to retrieve and copy the parser state if it has not changed since the last time it was saved. As the aspect only relies on the aspect interface pointcut `Syntax::rule_call()`, it is automatically open for future extensions, namely aspects that extend the syntax classes by additional rules.

The `SyntaxState` aspect is very closely connected with the syntax classes. Other aspects and classes regard the syntax classes and the `SyntaxState` aspects as a union. Therefore, it could be regarded as a *local* aspect. Nevertheless, it implements a highly crosscutting concern. In the C and C++ syntax it matches 104 and 118 grammar rules, respectively.

3.3 Syntax Tree Construction

In PUMA, syntax tree construction is implemented by the two *extension aspects* `CBuilder` and `CCBuilder` in combination with an *upcall aspect* `Builder`. The extension aspects introduce a function called `build()` into all classes that represent grammar rules. The implementation of these functions is different for each rule, because PUMA uses different C++ classes to represent the syntax tree nodes. If we wanted to change this in order to perform the syntax tree construction with a more generic aspect, this would merely require to replace the `CBuilder` and `CCBuilder` extension aspects. Figure 4 gives an overview of this design.

The `Builder` aspect is fully generic. It could even be used with syntax implementations of other languages than C or C++. It only depends on the assumption that each class that represents a grammar rule contains a static member function `build()`. This function is called after each successful run of the corresponding `parse()` function. The builder slices are higher-level code in the sense that they are aware of the implementation of the syntax classes, but not vice versa. Therefore, the `Builder` aspect falls into the *upcall aspect* category. The `build()` functions are introduced by the two *extension aspects* `CBuilder` and `CCBuilder`. If the `build()` function returns a syntax tree and not `NULL`, a pointer to this tree is pushed onto a stack, which has been introduced by the `Builder` aspect into the `Syntax` class. At the same time, the syntax trees that were pushed onto the stack by successfully parsed sub-rules are removed from the stack.

4. RELATED WORK

Several researchers have explored the benefits of AOP for compiler developments: De Moor and colleagues wrote a paper on “Aspect-Oriented Compilers” in 1999 [11]. However, while we concentrate on the parser front-end, their paper focuses on the semantic analysis with attribute grammars and the translation. Wu and associates describe a few AspectJ idioms for compiler construction [14]; however, their work also assumes that there is already a syntax tree. Also related is the design of aspect-oriented compiler construction systems called JustAdd [4]. In contrast to our design, which is centered around the grammar rules – which we regard as very stable! – JustAdd is centered around the syntax tree classes.

A very different direction has been taken by the very popular C++-based parser generator *spirit*[2], which uses C++ template metaprogramming to generate the parser. Although the grammar rules are written in C++, the description language looks like a DSL. However, there is no C++ grammar implementation for *spirit*, yet.

Somewhat comparable to PUMA is DMS by Semantic Designs [2, 1], a generic code transformation system that supports a number of target languages including C and C++. However, the source code of DMS is not available; hence, not many details about its parsing process are known.

5. CONCLUSIONS AND FUTURE WORK

For the development of PUMA, aspect-oriented software development and AspectC++ have worked well. Even though C++ is one of the most complex programming languages, the parser is still manageable. We achieved our key goals, which are configurability and extensibility. Both properties are needed to develop client-specific code analysis tools in very short time. Compared to GNU gcc/g++ the source code is quite small. The design is open for various kinds of extensions.

Concerning the future of PUMA we are very optimistic. It is the best open source C++ code analysis and transformation framework that we are aware of. At the moment the project is still a bit hidden (the source code is part of AspectC++), but in the future we plan to promote it more actively.

A challenging test case for the flexibility of our design will be the integration of the next C++ standard (C++1X), which is currently being finalized by the C++ standard committee. Most probably introduced by this standard will be the new `static_assert` keyword and feature [7], which provides for compile-time assertions in C++. The feature proposal for static assertions also contains an interesting statement about the estimated time for the integration of this feature into an existing compiler:

> *"A compiler writer could certainly implement this feature, as specified, in two or three days ..."*

Our integration into PUMA took us only a single day including tests, documentation, and some additional effort, which was needed, because it was the first C++1X feature that we integrated. Even though we are aware that the integration of *all* C++1X features will be a tremendous effort, we are optimistic that PUMA is well prepared for the upcoming requirements.

6. REFERENCES

[1] Robert L. Akers, Ira D. Baxter, Michael Mehlich, Brian J. Ellis, and Kenn R. Luecke. Reengineering c++ component models via automatic program transformation. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Ira D. Baxter. Design maintenance systems. *Communications of the ACM*, 35(4):73–89, 1992.

[3] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* Addison-Wesley, May 2000.

[4] Görel Hedin and Eva Magnusson. Jastadd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[5] The British Standards Institute. *The C++ Standard (Incorporating Technical Corrigendum No. 1)*. John Wiley & Sons, Inc., second edition, 2003. Printed version of the ISO/IEC 14882:2003 standard.

[6] ISO. The ansi c standard (c99). Technical Report WG14 N1124, ISO/IEC, 1999.

[7] Robert Klarer, John Maddock, Beman Dawes, and Howard Hinnant. Proposal to add static assertions to the core language (revision 3). Technical Report SC22/WG21/N1720, ISO/IEC, October 2004.

[8] Daniel Lohmann. *Aspect Awareness in the Development of Configurable System Software*. PhD thesis, Friedrich-Alexander University Erlangen-Nuremberg, 2009.

[9] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX Association.

[10] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Lean and efficient system software product lines: Where aspects beat objects. In Awais Rashid and Mehmet Aksit, editors, *Transactions on AOSD II*, number 4242 in Lecture Notes in Computer Science, pages 227–255. Springer-Verlag, 2006.

[11] Oege de Moor, Simon L. Peyton Jones, and Eric Van Wyk. Aspect-oriented compilers. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 121–133, London, UK, 2000. Springer-Verlag.

[12] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007.

[13] The Design Automation Standards Committee of the IEEE Computer Society. *IEEE Standard 1666-2005: SystemC*. IEEE, 2005.

[14] Xiaoqing Wu, Barrett R. Bryant, Jeff Gray, Suman Roychoudhury, and Marjan Mernik. Separation of concerns in compiler development using aspect-orientation. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1585–1590, New York, NY, USA, 2006. ACM.

[2] `http://spirit.sourceforge.net/`