Implementing concern-specific languages with abc

Seminar on Aspect-oriented programming
Institut für Informationssysteme Fachgebiet Wissensbasierte Systeme
Hannover University

Eric Bodden*

February 2005

Abstract

In this work first we introduce the notion of concern specific languages (CSL) [Bod05] which are to a specific crosscutting concern, what domain specific languages are to a specific domain. Implementing such CSLs was a tedious task in the past since no extensible frameworks for implementing crosscutting concerns existed. Ostermann and Mezini [OM05] proposed a Prolog based implementation of aspect-oriented programming which would enable easy extension of the programming language. However, it is not yet clear what runtime impact will be involved with such an approach. With the AspectBench Compiler (abc) [ACH$^+$05], which was released in October 2004, researchers now have a powerful extensible compiler for the aspect-oriented language AspectJ, enabling easy implementation of language extensions or even whole CSL for a specific crosscutting concern. We first motivate CSLs in general and then introduce the abc framework. In the subsequent chapters we introduce our specific CSL and report on the steps necessary to implementing it using abc. Finally we recapitulate on the ease of use of abc and conclude with a proposal for further development of this compiler framework.

1 Concern specific languages

Through the advances in model and domain driven development in combination with more extensible compiler frameworks and code generation technologies, we have seen a rise of domain specific languages (DSL) for the most different applications during the last decade. Once given an implementation for a DSL, such a language is usually easier to understand and maintain with respect to the problem it solves [vDKV00]. DSLs have the advantage that they usually use a notation that comes close to notations that people are used to who are concerned with the problem domain. This enables a straightforward translation from specification and models into the DSL and with such into a running implementation.

*Eric Bodden is currently commencing his diploma thesis at RWTH Aachen University

Another upcoming tendency in software engineering during the last years was and still is the wish to separate crosscutting concerns. Such concerns typically consist of code which functionally builds a single unit but is though usually scattered throughout the whole application due to limitations of the implementation language. Aspect-oriented programming tries to overcome this problem by separating such concerns into single units called *aspects* and has proven very powerful in a lot of applications domains in the past.

Though, implementation of such aspects can still be very tedious as long as implementors are bound to an aspect-oriented programming language such as AspectJ. In the same way DSLs aid the development process of software for a certain domain, CSL can here aid the development process for a single concern. As an example, in this paper, we will consider the concern of Runtime Verification (RV), which is in general applicable to any problem domain but always implements the same functionality: To check if a program satisfies certain formalized conditions at runtime. This concern is usually not only scattered through the whole application (RV could be applied to any part of the system) but also exists in basically any software development process. Thus a language that would ease implementation of this concern would not be domain specific but rather specific to this single concern.

It is only natural that aspect-oriented languages such as AspectJ facilitate the implementation of CSLs since AOP enables modular implementation of crosscutting concerns. The implementation of a specific CSL can thus be performed by reducing input in the CSL to aspect-oriented code in AspectJ by a preprocessing approach rewriting the abstract syntax tree in the CSL to an equivalent syntax tree holding only AspectJ constructs.

In this work we demonstrate how the open compiler framework AspectBench (abc) [ACH$^+$05] can be used in order to implement in particular CSLs, as well as AspectJ language extensions, through code generation. abc is an extensible compiler for the aspect-oriented language AspectJ [KHH$^+$01], which supports modular implementation of so-called crosscutting concerns. In addition to usual Java, exploiting the support for crosscutting concerns, which AspectJ provides, we show, that using abc, one can easily implement CSLs which are very expressive, since they can reason about entire sets of events in the execution flow of a program and thus implement a whole concern in a single aspect unit.

As an example we define a linear-time temporal logic (LTL) over pointcuts, which is usually the language of choice for reasoning about paths, here specifically about runtime execution paths. Thus it is mostly the natural language for this concern. People that are meant to implement this concern will find writing code in LTL much easier than writing the appropriate aspect that actually implements the functionality. The proposed CSL enables the user to annotate given source code with temporal formulas that should be verified during program execution. We extend abc to implement this language by reducing specifications in the CSL to code in the AspectJ language, which in turn is translated to usual Java code using the abc core implementation. The aforementioned annotations are implemented as annotation types, which were introduced in Java 5 [JSR]. They are extracted employing the bytecode analysis kit BAT[1]. This is not shipped with abc. An implementation of the annotation extraction code is available form the author on request.

[1]see http://www.st.informatik.tu-darmstadt.de/

2 AOP: History and Introduction

AspectJ is today the most widely used aspect-oriented programming (AOP) language. It was originally developed by Xerox PARC[2]in the late 90's. Furtheron various companies and researchers contributed to its development. Particularly IBM keeps pushing forward AspectJ till this date and provides powerful tool integration for several IDEs, especially Eclipse[3] which originated from IBM. They are also using AOP excessively today in a production environment for their application middleware products.

The purpose of AOP is to separate crosscutting concerns. Such concerns are typically technical features that scatter throughout a given program and whose implementation is usually not part of the application core. Though there have been several workshops on aspect mining[4] in the past, some people suggest that aspects do mostly not occur as natural roles when specifying applications but rather tend to capture technical implementation details [Ste04].

AOP can generally be applied to any language. Functional languages like LISP and SCHEME [CI91] tend to have support for AOP almost builtin [Cos03], and indeed the idea of AOP, as one of the major inventors Gregor Kiczales mentions [Lem04], originates from experiences with MacLisp. For instance the notion of a piece of *advice* stems from advice in this language. A *piece of advice* is a piece of source code that implements some crosscutting functionality. An *aspect* in the AspectJ language comprises such pieces of advice plus a set of so-called *pointcuts*. A pointcut is a certain kind of regular expression that is able to describe and pick out points in the runtime control flow of the core application (so-called *joinpoints*).

In AspectJ one is able to build such expressions that reason about the static structure (e.g. every point within a certain class) as well as the execution flow (e.g. all calls to a certain method within the control flow of another). This is usually powerful enough to refactor out most crosscutting concerns from a given business application.

Table 1: AspectJ pointcut and advice logging authentication events

```
pointcut auth(User u):
  call(* Authentication.login(User)) && args(u);

after returning(): auth(User user) {
 SecurityLog.log("User " + user.getId() + " logged in");
}
```

However there are often applications where reasoning about other properties of a program is necessary, for instance certain conditions over traces of the execution flow. Walker et al. [WV04] implemented so-called *tracecuts*, certain pointcuts that let the user specify context-free expressions over the execution flow, which have been matched in order to make such a tracecut apply to a

[2]Palo Alto Research Center

[3]http://www.eclipse.org/

[4]Aspect mining describes the process of revealing crosscutting concerns in a given specification or application.

certain joinpoint. Südholt et al. [DFS05] follow a similar approach using event-based AOP. The AOP implementation JAsCO [SVJ03] enables a simple notation of stateful aspects [DFS04]. Sakurai et al. [SMU+04] propose association aspects which can be used to easily implement and establish relations between arbitrary objects in an application. Ostermann and Mezini [OM05] propose pointcuts holding expressions in the Prolog logical programming language and are even more powerful (Turing complete).

In a related work [Bod05], we describe the power of such languages, give some examples and show how CSLs can be composed and facilitate each other's implementation. As it turns out, two of those CSLs, association aspects and stateful aspects, could indeed have facilitated the implementation we describe here. We include an example for each of the two CSLs on page 5.

So in general one can conclude that there seems to be a clear interest in more expressive crosscutting languages and as a consequence in concern specific languages as well.

3 Why using abc?

Unfortunately in the past, such proposed language extensions have all gone into different builds of various compilers - mostly into the ajc [HH04] compiler (the original implementation by PARC) but also into others like JAsCo [SVJ03], AspectWerkz [Bon04] or in the form of hand coded preprocessors. The AspectBench Compiler which was developed by McGill and Oxford now facilitates such extensions by providing an extensible, optimizing compiler for the AspectJ programming language. This will enable researchers henceforth to implement and/or port such extensions into one common framework and so reuse their implementations at once [Bod05].

In this paper we demonstrate how concern-specific languages can be implemented using abc. As one will see, employing abc for this purpose has several benefits over other approaches: On the one hand abc is an open compiler framework that is easy to extend. On the other hand, some language extensions are already implemented using abc and thus can be used by other extenders at once. We will make use of this mechanism by reducing free variables in our specific language extension to so-called *pointcut-private* variables, which are part of the EAJ extensions, described in [ACH+05].

An advantage of true CSLs over other language extensions is that they are usually orthogonal to the (Java) base implementation. That leads to the fact that in such cases simple rewrites of the abstract syntax tree (AST) suffice for the language implementation (resulting in a plain AspectJ AST), since their type system and semantic checks do not interfere with the implementations that exist already for Java respectively AspectJ. Thus, the CSL implementation exists really as an addition, not a replacement to existing code.

In the following chapters we will point out, how this can ease the development process. In particular one consequence is that we do not need to touch any of the analyses and transformations which are used to implement AspectJ, since we are not extending AspectJ itself, but rather reducing to it. The type system and appropriate semantic checks do normally not have to be touched.

However first we will give a brief overview about the structure of abc.

Table 2: Stateful aspects in JAsCO

```
start>p1;
p1: execute(startmethod) > p3||p2;
p3: execute(stopmethod) > p1;
p2: execute(runningmethod) > p3||p2;

after() p2() {
  //do something
}
```

This specifies a start state p1, from which a transition to p3 or p2 can be taken by *execute(startmethod)* and so forth. A piece of advice can be bound to each state.

Table 3: Association aspects by Sakurai et al.

```
aspect Equality perobjects(Bit, Bit) {
  Bit left, right;
  Equality(Bit l, Bit r) {
    associate(l, r);     //establishes
    left = l; right = r; //association
  }
  after(Bit l) : call(void Bit.set())
    && target(l) && associated(l,*){
    propagateSet(right); //when left is called,
  }                      //call set on right
  after(Bit r) : call(void Bit.set())
  && target(r) && associated(*,r){
    propagateSet(left); //when right is called,
  }                     //call set on left
  //helper methods go here
}
```

This association aspect implements an enforced equality relation by associating an aspect instance which each two Bit objects which should be equal. On a call to `set()` on each such bit, the value is propagated to the other bit at once. Such a relation can be instantiated explicitly using

```
Bit b1 = new Bit(), b2 = new Bit();
Equality a1 = new Equality(b1,b2);
```

4 Structure of abc

The major Java based compilers for AOP languages that are around today, are all so-called *weaving compilers*: They have two major passes, one compilation pass, where the aspects are translated into Java bytecode using a special compiler for that language, and one weaving pass, where calls to the appropriate pieces of advice are woven into the actual core application at all the places where pointcuts apply. Runtime checks are inserted at all the necessary places.

As such a compiler, abc is based on two major frameworks: As compiler frontend the Polyglot [NCM03] compiler toolkit is used. Polyglot is a compiler framework built as front-end to PPG, an extensible LALR parser generator based on the CUP LALR parser generator for Java. In PPG, existing grammars can optionally be extended by *extending* or *dropping* productions of a base grammar. Also, Polyglot uses object association in favor over class inheritance employing a sophisticated delegation model. This allows extenders to add or replace functionality piece by piece to distinct node types of the abstract syntax tree which do not need to share common super types.

As the weaving backend, the bytecode analysis and optimization framework Soot is being employed. Soot is able to load Polyglot ASTs and/or Java bytecode and transform those into an internal three address code representation called *Jimple*. This representation is stackless and as such allows for relatively easy code transformations and analyses. The weaving process, that implements the translation from AspectJ into plain Java, makes use of this representation. Since Soot is also an optimization framework, many intra- and interprocedural analyses are already builtin and can easily be extended. They can be applied to the readily woven code at once, thus generating more efficient code than ajc does, in certain situations. With respect to compile time performance, however abc tends to be slower than ajc due to it's heavily object-oriented structure. Whereas ajc is optimized for compile time performance, abc is optimized for extensibility and run time performance of the resulting bytecode.

4.1 Polyglot

Polyglot as the abc compiler frontend, facilitates easy extendability in several dimensions. This is an enormous benefit over earlier approaches in compiler technologies, which usually only allowed extendability by the means of class inheritance, which is truly one-dimensional: Each AST node inherits functionality from its parent nodes and from nowhere else. During the last years however, many authors like Gamma et al. have suggested to use object composition in favor over class inheritance, because it tends to lead to more flexible system designs (see [GHJV95], pp. 18-20). Polyglot makes consequent use of the delegation pattern, that allows for such object composition:

Each AST node, whenever visited, dispatches this message first to its delegate object, which by default is the visited object itself.

Using this mechanism, one can easily *replace* or extend functionality that is spread over various node types, which do not need to share common super types.

In addition to delegates, nodes also support a chain of extension objects. An extension is meant to *add members* to a set of node types.

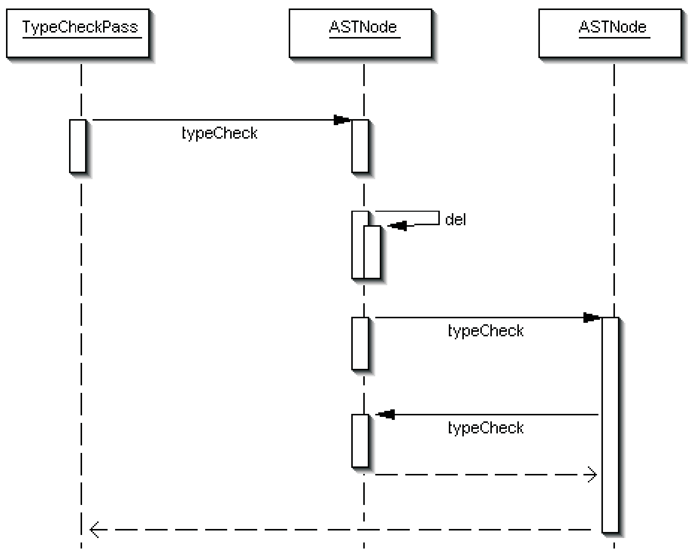

Figure 1: Polyglot delegation model (with call back to original receiver)

Polyglot also supports type checking and other semantic passes for the Java language. However since we are doing a source to source transformation, we are not going to extend those facilities. We only make use of them implicitly through the final transformation processes to Java bytecode.

4.2 Soot

Soot is a bytecode analysis and optimization framework, which provides common templates for inter- and intraprocedural analyses. Several such analyses are already builtin. They comprise even complex *points-to* and *flow* analyses, which can be used to reason about control flow, possible method dispatches at runtime and so forth. Obviously, by making use of information produced by such static analyses, an AspectJ compiler can generate much more efficient code under certain circumstances. For instance, the evaluation of *cflow* could be dramatically accelerated by replacing stacks with counters, which is possible in most common situations [DGH$^+$04].

Nevertheless, Soot is, in the first place, used within abc because of the *Jimple* representation it provides. A Jimple program consists of a stackless, three-address code[5] representation of Java bytecode. In Jimple, all implicit method invocations (e.g. String concatenation) and implicit references to the current object (`this`) have been resolved. As a result, all objects that contribute to the implementation of a method body are explicitly available in a local variable and each statement consists only of at most one method call and one assignment. This makes Jimple easy to process and an ideal base for modifications as they have to be performed by the advice weaving process. Table 4 on page 8 gives an example of this representation.

[5]Object, arguments and result

Table 4: Java class and corresponding Jimple code

```
public class Foo {
    int a;
    public int f(int x,int y , int z) {
        return a+x*y+z;
    }
}
public class Foo extends java.lang.Object
{
    int a;

    public int f(int, int, int)
    {
        Foo this;
        int x, y, z, $i0, $i1, $i2, $i3;

        this := @this: Foo;
        x := @parameter0: int;
        y := @parameter1: int;
        z := @parameter2: int;
        $i0 = this.<Foo: int a>;
        $i1 = x * y;
        $i2 = $i0 + $i1;
        $i3 = $i2 + z;
        return $i3;
    }

    [Implicit constructor omitted]
}
```

Thus, weaving is implemented by generating a so-called *AspectInfo* data structure, which describes transformations on the level of Jimple code. This code can then, using Soot, be transformed to bytecode or source code again. The latter is particularly useful for educational purposes, since one can see at once, how advice weaving affects given classes.

5 The proposed language extension

As concern-specific language, we propose an extension to the original Java language (first introduced in [Bod04]), that implements runtime checks of temporal formulas enabling developers to reason about the execution trace of a program. With respect to the original Java language, our extension is orthogonal, since it does not interfere with the original implementation of the compilation: Just additional code is added to implement the necessary runtime checks, which trigger tracking of the stated formulas. This is transparent to the core application - it's behavior is not modified in any way.

The syntax of our language is as follows.

5.1 Syntax and semantics

We allow the temporal operators of next-time free linear-time temporal logic (LTL) [Pnu77], which are F, G and U. Those can be cascaded and take usual AspectJ pointcuts as propositions.

For any given formulas or propositions φ and ψ, we define the semantics of G, F, and U at a given joinpoint t as follows:

- $G(\varphi)$ is true at t iff *Globally* on the path from the start of the application up to t, φ was true.
  Example: `G( if(User.loggedIn()) || !call(* Account.debit()) )`
  Globally, either the user is logged in or no call to Account.debit() happens.

- $F(\varphi)$ is true at t iff *Finally* somewhere on the path from the start of the application up to t, φ was true.
  Example: `F( execution(* Locking.releaseAllLocks()) )`
  All locks are finally released (by executing Locking.releaseAllLocks()).

- $(\varphi U \psi)$ is true at t iff on the path from the start of the application up to t, φ was true *Until* finally ψ became true (before t):
  $(\varphi U \psi)_t = \psi \vee (\varphi \wedge (\varphi U \psi)_{t'})$ for t' being the next joinpoint after t in the current control flow. Either ψ is true (release) or φ is (still) true and the whole formula holds on the subsequent path.
  Example:
  ```
  (!call(* User.login(Credentials))) U
            (if(Time.getTime() > User.lastLogin() + 60000))
  ```
  We allow no call to User.login(Credentials) until at least 60 seconds after the last login attempt.

In addition, we allow composition of those constructs as well as free variables in such formulas, which are internally bound by the contained pointcuts. In the following example, `o1`, `o2` and `bool` are such variables. Formulas are put into an `LTL` term constructor to allow for easier parsing.

Table 5: Example using variables: (meaning *Object.equals(Object) is reflexive)*

```
LTL(
Object o1,o2; boolean bool:
  G (
      (call(boolean Object.equals(Object)) returning bool
        && target(o1) && args(o2)) ->
      (o1!=o2 || bool == true)
    )
)
```

5.2 Code generation outline

Since, as mentioned above, this CSL is mostly orthogonal to the Java base code, we can implement it by simply rewriting the AST, that is generated by Polyglot

Table 6: Example composing temporal operators (meaning *all acquired locks are finally being released*)

```
LTL(
Lock l:
  G ( !call(* Lock.acquire(l)) || F ( call(* Lock.release(l)) ) )
)
```

into a plain AspectJ AST. This is then automatically rewritten again into a pure Java AST by the original abc implementation in subsequent compiler passes. See chapter 6.5 for details.

6 Implementing the language extension

In fact, AspectJ is - in abc - not more than a language extension to Java itself. Thus, in order to implement our CSL, we simply need to inject appropriate AST rewriting passes into the chain that abc uses for its own AST transformations. This section gives some general directions on how abc should be extended in order to implement a CSL and what we had to do in our specific example.

In general, an extension to abc consists of a single package (here `abc.ltl`) with the following subpackages and files:

- `ast` holds classes for AST nodes of the CSL or language extension. (see chapter 6.3 on page 13)
- `extension` holds classes for node extension objects as described above.
- `parse` holds JFlex lexer and PPG grammar definitions.
- `types` holds classes building the type system of the CSL or extension.
- `visit` holds visitor classes implementing the AST rewriting passes.
- `abcExtension.java` instantiates the extension using a factory method, optionally adding lexer keywords and specifying a factory for creating reflective joinpoint objects at runtime.
- `ExtensionInfo.java` defines the actual extension instantiating the appropriate lexer and parser and arranging the AST rewriting passes.

One also needs to customize the Ant build script, which invokes JFlex, CUP, PPG, an XSLT transformer and Javadoc in order to generate lexer, parser, a command line options parser (using XSLT) and optionally the documentation for the runtime libraries that allow reflective access to AspectJ joinpoints. As a result of the build process one receives the readily packaged JAR file *lib/abc.jar*. In order to run abc with the custom language extension, it is being invoked using the `-ext` parameter, here `-ext abc.ltl`. This instantiates the appropriate extension using the factory method `abcExtension.makeExtensionInfo`.

In order to include a custom extension into the build file, one can simply have a look at how the default extension *eaj* was integrated and deal with the new extension in the very same way.

6.1 Scanner/Lexer

Polyglot in general uses a lexer generated by JFlex. If implementing a CSL which has nothing in common with plain Java code, you may just want to generate your own custom lexer.

However if you are extending Java or AspectJ with additional functionality as we are, you may want to reuse the lexer that comes with abc. abc provides a JFlex input file from which a lexer for AspectJ (which comprises plain Java) is generated. This lexer can easily be parameterized at runtime, to allow additional keywords. If you need additional functionality, you might need to extend, rewrite or replace the abc lexer, though. However, future releases of abc will likely be providing more extensibility here.

For our approach, it suffices to add keywords to the appropriate lexer states. This can be done by overriding the appropriate method of `abcExtension`:

```
public void initLexerKeywords(AbcLexer lexer) {
    // Add the base keywords
    super.initLexerKeywords(lexer);

    lexer.addAspectJKeyword("LTL",
      new LexerAction_c(
        new Integer(abc.ltl.parse.sym.LTL_DECL_PREFIX),
        //switch to pointcut state when scanning this keyword
        new Integer(lexer.pointcut_state()))
      );

    lexer.addPointcutKeyword("F",
      new LexerAction_c(
        new Integer(abc.ltl.parse.sym.LTL_FINALLY))
      );

    lexer.addPointcutKeyword("G",
      new LexerAction_c(
        new Integer(abc.ltl.parse.sym.LTL_GLOBALLY))
      );

    lexer.addPointcutKeyword("U",
      new LexerAction_c(
        new Integer(abc.ltl.parse.sym.LTL_UNTIL))
      );
}
```

Note that the token constants `abc.ltl.parse.sym.*` are generated by the PPG parser generator and so they *match* the appropriate parser implementation.

6.2 Parser

The parser consumes those tokens with the productions defined in its grammar, optionally building an AST in its action rules. Tokens are defined by defining the appropriate terminal symbols in the PPG grammar. Here:

```
terminal Token LTL_DECL_PREFIX;  //LTL
terminal Token LTL_FINALLY;      //F
terminal Token LTL_GLOBALLY;     //G
terminal Token LTL_UNTIL;        //U
```

If your CSL extends Java or AspectJ as here, you might want to make use of PPG's grammar extension facility: Just include the Java respectively AspectJ base grammar. Here we use `include "../../aspectj/parse/aspectj.ppg"`. Existing productions can then be altered using `drop` and `extend` keywords.

In any case you will have to define the new non-terminal symbols and productions. The former may be typed. In that case an object of that type may be generated by the appropriate production rule and returned using the `RESULT` variable. The abstract syntax tree is so built automatically by cascading those result objects.

Table 7: Extract of the grammar for the proposed language extension

```
//The whole LTL formula pointcut.
non terminal PCLTLGeneral ltl_formula;
//The optional private variable list.
non terminal List ltl_formula_private_var_decl_opt;
//The formula body.
non terminal PCLTLGeneral ltl_formula_body;

//ltl formula body
ltl_formula_body ::=
    //LTL operators with inner pointcuts
    LTL_FINALLY:x LPAREN pointcut_expr:a RPAREN:y
    {:
      //ask the factory to create a PCLTLFinally node
      RESULT = parser.nf.PCLTLFinally(parser.pos(x,y),a);
    :}
  |
    LTL_GLOBALLY:x LPAREN pointcut_expr:a RPAREN:y
    {:
      //ask the factory to create a PCLTLGlobally node
      RESULT = parser.nf.PCLTLGlobally(parser.pos(x,y),a);
    :}
  |
    LPAREN:x pointcut_expr:l RPAREN LTL_UNTIL
    LPAREN pointcut_expr:r RPAREN:y
    {:
      //ask the factory to create a PCLTLUntil node
      RESULT = parser.nf.PCLTLUntil(parser.pos(x,y),l,r);
    :}
;
```

Note that child AST nodes may be referred to using labels (here `a`,`x`,`y`,`l`,`r`). also nodes are not instantiated directly but rather using an associated Node factory `nf` (see chapter 6.4 for details). This has the advantage, that further extensions can easily replace nodes by custom implementations by simply overriding the appropriate factory methods.

For debugging purposes, we recommend to set `parserTraceOn = true;` in the constructor of your parser. This will produce a parse trace. Messages can be passed to the trace using the `parseTrace(String)` method. The classes `Debug` and `Report` allow tracing various rewriting passes.

6.3 AST nodes

The AST nodes themselves have to be written in accordance to the AST node types that have been stated in the above grammar. Those nodes will be instantiated by the appropriate node factory. All nodes have to adhere to the delegation model as described in chapter 4.1. In addition, they (and their optionally associated node extensions) implement major functionality to support AST rewriting passes.

For each implementation of a node type (whose name usually ends on *_c*), an appropriate interface type must be given, which exposes those methods that should be visible to the parser. The parser must never be aware of any implementation details. Thus only interface types must be used in the parser implementation. The connection between those interfaces and their implementations is performed by the node factory.

6.4 AST node factory

The node factory instantiates new AST nodes of the appropriate implementing classes for a given interface. Thus, the factory must hold a method for each constructor of each AST node, that may be created by the parser, e.g. for the node for the *finally* pointcut:

```
public PCLTLFinally PCLTLFinally(Position pos, Pointcut pc) {
  return new PCLTLFinally_c(pos,pc);
}
```

By employing this mechanism, node types can later on easily be exchanged by simply extending the node factory. Delegates and extensions can be added to a node by simply setting its extension using the appropriate `del` and `ext` methods (see [NCM03]).

6.5 Visitor passes

The AST passes implement the major code transformation and generation facility. Such passes can be of different nature. Usually they rewrite the AST as *visitor* ([GHJV95]) to implement the actual code transformation. However there are also passes that implement semantic checks or dump code as well as so-called *barrier passes* which synchronize rewriting passes. A barrier pass ensures that all jobs[6] of this pass have at least reached this barrier.

Each rewriting pass implements a visitor, looking out for AST nodes it affects, and then dispatches the methods actually implementing the pass on the node object and/or its extension objects.

[6] A *job* in Polyglot reflects basically a compilation unit in its currently rewritten state.

Partial ASTs can also be generated using Polyglots quasiquoting facilities: Using the class `QQ`, one can convert Java code in String format into readily parsed partial ASTs. Unfortunately this powerful tool is only available for the base implementation (Java) at the moment, neither for AspectJ nor for any other extension. Using such partial ASTs, code generation could by eased a lot, because AST rewrites could be simulated by simple String rewrites in many cases.

Generally, passes are scheduled by overriding `ExtensionInfo.passes(Job)`. For implementing a CSL, one should first call the appropriate parsing (and optionally type checking) passes of abc, then schedule the transformation passes which convert the AST of the CSL into a plain AspectJ or Java AST and finally schedule the passes that convert AspectJ into Java and write the resulting Java code to disk.

For our specific example of LTL we employ the following passes:

- Parse the annotated bytecode using the BAT bytecode toolkit, extracting the LTL formulas. Generate initial `SyntheticSource` (see below) instances holding only those formulas, e.g.:

  ```
  public aspect F302749537 {
      F(staticinitialization(SomeClass))
  }
  ```

  The generated sources are injected into the compilation process as follows:

  - We derived a new class `SyntheticSource` from `FileSource`. The latter comes with Polyglot and represents a source File in the file system. Thus its major task is to provide an input stream to the content of the associated file. Since we generate units on the fly without disk access, `SyntheticSource` generates this stream from a byte array that holds the class definition in String format.
  - Those synthetic sources were then passed to the framework using `ExtensionInfo.addJob(Source)` (still before the first pass is scheduled to run, but after the compiler is initialized - we found the method `ExtensionInfo.initCompiler(Compiler)` to be the right place, after the call to `super`).

- Parse those sources to generate an AST comprising the AST nodes for the contained formulas. For this purpose, we reuse the given method `ExtensionInfo.passes_parse_and_clean()`.

- Resolve all types for variables contained in those formulas. The passes added by `ExtensionInfo.passes_disambiguate_signatures()` can be reused for this purpose. Also the corresponding new AST nodes need to override the appropriate disambiguation methods, this pass delegates to.

- Bring the formulas into Disjoint Normal Form (DNF). This is necessary, since each conjunct represents one primitive pointcut which describes a single set of joinpoints to match. abc already provides a class DNF for this purpose in the appropriate `Pointcut` class in the `weavinginfo` package. *Weaving infos* represent the implementation strategy that is associated with a certain AspectJ construct. Again we override the appropriate implementing methods in the new weaving info nodes to not only convert the contained AspectJ pointcuts but the whole formulas including LTL operators in DNF.

- Transform the AST for the formula into an AST holding a piece of advice for every conjunct. This advice switches the aspect-internal state whenever an appropriate joinpoint is matched on the execution trace at runtime. Formula-private variables are partially translated to aspect instance variables and partly to pointcut-private variables provided by the *eaj* extension that comes with abc.

- Generate additional inter-type declarations on types for objects that are to be tracked (see example on page 9) in order to indicate that a certain formula holds *for this particular object*. They track, which pointcut conjucts already matched on the associated object.

- Generate some additional code that keeps track of the implementing aspects and shows results during the later program execution.

6.6 Work flow of the extended abc implementation

With the now implemented compiler for the given CSL, one invokes abc using `-ext abc.ltl`. This instantiates the appropriate `ExtensionInfo` as well as the connected lexer and parser, and finally schedules and executes the AST passes previously arranged. As output one receives plain Java source code or bytecode implementing the concern specific language.

The following section gives a quick glance at what the implemented code transformations produce.

6.7 Example transformation

The following formula models that for a given web shop it should not happen that a purchase is being tried although the shop was put into standby mode (which might happen for maintenance activities) or shut down[7]:

```
G(
  !( (call(* WebShop.standby()) || call(* WebShop.shutdown()))
     && F( call(* Webshop.purchaseItem(Item)) )
   )
)
```

Translation into AspectJ leads to the following aspect:

```
public aspect G123 {

  boolean matched_pc1, matched_pc2, matched_pc3;

  pointcut pc1(): call(* WebShop.standby());
  pointcut pc2(): call(* WebShop.shutdown());
  pointcut pc3(): call(* WebShop.purchaseItem(Item));
```

[7]The example omits checks on re-enabling of the web shop for brevity. We will very likely provide macros for easy formulations of formulas as *Call to* a *but not yet to* b.

```
  after(): pc1() {
   matched_pc1 = true;
  }

  after(): pc2() {
   matched_pc2 = true;
  }

  after(): pc3() && if(matched_pc1 || matched_pc2) {
  //if pc1 or pc2 were seen previously, match pc3
   matched_pc3 = true;
  }

  boolean globalCheck() {
  //implementation of the outermost check
   return !( (matched_pc1 && matched_pc3)
          || (matched_pc2 && matched_pc3));
  }

  after(): set(matched_pc1) || set(matched_pc2)
      || set(matched_pc3) {
  //if any state is changed, apply the check
   if(!globalCheck()) {
      RuntimeVerifier.reportError(this);
   }
  }

}
```

After this step, the AspectJ weaver, which is integrated into abc instruments the core implementation with the aspect stated above.

In the following we would like to recapitulate and conclude with an evaluation of the abc architecture.

7 Experience report

7.1 Ease of use

When starting our implementation we first came forward very quickly. Writing a skeleton extension and linking it using the Ant build script took not more than an hour. Extending the lexer and grammar was also quite straightforward. The parsing passes of abc could be completely reused in order to gain an initial AST in our CSL.

When implementing the actual transformations, we found the sophisticated delegation model to be the initial challenge which abc extenders have to overcome. In general the delegation model provides for scalable extensibility and good modularity, however we found that it might take some time to get the implementation right.

With regard to the aspect-oriented extensions that abc adds to the Polyglot base compiler, we found that little documentation is available at the current time about the contained compiler passes and especially the generation and consumption of the *AspectInfo* which states how pieces of advice are woven into the core application. Some high level information is available in [ACH$^+$05], however this document does not go deep enough to completely reflect the code generation process. Also the code comments leave quite some room for improvements at the moment.

In general, however, we conclude that abc is a very powerful framework, making implementation of CSLs and language extensions to AspectJ a rather straightforward task. Code reuse is indeed maximized as far as possible through the delegation model. Parsers can easily be generated using PPG, even extending existing grammars in a convenient way. Thus extenders can concentrate on the actual translation of their CSL into plain AspectJ code. Once the AST was rewritten to a plain AspectJ AST, the actual source code or bytecode generation can simply be delegated to abc. This accelerates the development process a lot.

It should be noted that a lot of this power already derives from the Polyglot base implementation. However, the crosscutting concerns, which may be implemented only in combination using abc, prove often very powerful - especially for CSLs.

7.2 Suggestions for abc extenders

For abc extenders we recommend in addition to this paper the publication [ACH$^+$05] which gives more high-level information about abc and about its extendability. Also we would like to encourage people extending abc to publish their extensions and inform other users about the abc-dev mailing list[8]. This enables others to reuse those language extensions and build their extensions on top of those. If all available AspectJ extensions, which are already available today had gone into a common framework as abc, this would certainly have made prototyping of new research ideas a lot easier [Bod05].

7.3 Suggestions for abc developers

First we wish to thank the abc team for providing this framework which made the job of implementing our CSL a lot easier than first expected. However, since abc is still in is first version, we see still room for improvement.

As stated above, code comments are often not available or very concise. Especially some more comments on the rewriting passes and how they interact with or depend on each other would be quite helpful.

The lexer is currently only extensible with additional keywords, which often suffices. However we think it would be desirable if it would support addition of states as well.

Also we found the quasiquoting facility of Polyglot to be a great feature, which is unfortunately only available for the base language (Java) at the moment. It would be very helpful if this facility was extended to support AspectJ constructs as well, or even better constructs of any possible abc extension. It might be helpful to compare to META-AspectJ (see below) [ZHS04].

[8]available at http://abc.comlab.ox.ac.uk/lists

Also it might be worthwhile looking into something suggested in [Wil05] : An XML based interface for intermediate code transformations. This would probably suffice for some language enhancements and would eliminate the immediate need to extend the actual abc framework at all. Simple XML transformations could be implemented in XSLT, XQuery or similar formalisms.

8 Related Work

To our best knowledge this is the first report about extending abc written by someone not being member of the abc development team. Also it seems to be the first report describing the implementation of a real concern specific language, in contrast to the EAJ extensions mentioned in [ACH+05], which implement rather small additions to the AspectJ syntax and semantics.

Related work regarding abc momentarily seems to be constrained to the tool META-AspectJ (MAJ) [ZHS04] by the Georgia Institute of Technology. MAJ can be seen as an AspectJ code generator with extended quasiquoting facilities. Its most powerful feature is a fully-automatic type inference algorithm: For virtually any syntactically correct AspectJ construct, given in String format, this construct can be parsed. The type of the root AST node for this construct is automatically inferred using a sophisticated algorithm with backtracking facilities. This feature heavily eases the development process since only one single interface is necessary to convert Strings to AST nodes and the developer does not even need to bother about the actual AST implementation. However, META-AspectJ is an AspectJ code generator and not more. This means in particular, that it provides no support at all for parsing a CSL or language extension, such as abc does. (It has no frontend.) Also it does not provide any weaving of AspectJ code to plain Java code, nor does it provide any of the static analyses available in abc through Soot (no backend). As mentioned above, quasiquoting facilities with type inference as in META-AspectJ could however be a valuable addition to abc in the future.

9 Conclusion

We hope this paper to be a valuable guide for implementors of concern specific languages or language extensions to AspectJ. We have described how to parse a CSL using abc, what steps are necessary to convert the AST in the CSL into a plain AspectJ AST and how to finally compile this AST into plain Java source or bytecode. We have pointed out major implementation details which are important to understand the design and internal processes of abc. Also we have given hints on how an implementation can be debugged and traced. We gave suggestions for future development.

We conclude that abc makes implementing such CSL or language extensions as straightforward as it can be and want to encourage others to try themselves and publish their extensions for further reuse and cooperation.

References

[ACH$^+$05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotk, Ondrj Lhotk, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. An extensible AspectJ compiler. In *Proceedings of the Fourth ACM SIG International Conference on Aspect-Oriented Software Development (AOSD'05)*, Chicago, USA, March 2005. ACM Press.

[Bod04] Eric Bodden. A lightweight LTL runtime verification tool for Java. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 306–307. ACM Press, 2004.

[Bod05] Eric Bodden. Concern specific languages and their implementation with abc. Download: http://www.bodden.de/publications, 2005.

[Bon04] Jonas Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In Murphy and Lieberherr [ML04], pages 5–6.

[CI91] Inc. Staff CORPORATE IEEE. *IEEE Std 1178-1990, IEEE Standard for the Scheme Programming Language*. IEEE Standards Office, 1991.

[Cos03] Pascal Costanza. Dynamically scoped functions as the essence of AOP. *SIGPLAN Notices*, 38(8):29–36, 2003.

[DFS04] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150. ACM Press, 2004.

[DFS05] Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005.

[DGH$^+$04] Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 150–169. ACM Press, 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In Murphy and Lieberherr [ML04], pages 26–35.

[JSR] Java specification request for standardized metadata annotations (JSR175). http://jcp.org/en/jsr/detail?id=175.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[Lem04] Otvio Augusto Lazzarini Lemos. Is 'advice' adequate?, 08 2004. Thread in the aosd-discuss mailing list (http://www.aosd.net/).

[ML04] Gail C. Murphy and Karl J. Lieberherr, editors. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD 2004, Lancaster, UK, March 22-24, 2004*. ACM, 2004.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *CC*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2003.

[OM05] Klaus Ostermann and Mira Mezini. Design and implementation of pointcuts over rich program models. *TechReport TU Darmstadt*, 2005. http://www.st.informatik.tu-darmstadt.de/.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[SMU+04] Kouhei Sakurai, Hidehiko Masuhara, Naoyasu Ubayashi, Saeko Matsuura, and Seiichi Komiya. Association aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 16–25. ACM Press, 2004.

[Ste04] Friedrich Steimann. Why most domain models are aspect free. In *5th Aspect-Oriented Modeling Workshop AOM/UML*, 2004.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *AOSD*, pages 21–29, 2003.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[Wil05] Gregory V. Wilson. Extensible programming for the 21st century. *ACM Queue*, 2(9):48–57, 2005.

[WV04] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In Richard N. Taylor and Matthew B. Dwyer, editors, *SIGSOFT FSE*, pages 159–169. ACM, 2004.

[ZHS04] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating AspectJ programs with meta-AspectJ. In *Generative Programming and Component Engineering (GPCE)*. Springer-Verlag, October 2004.