Fine-Grained Generic Aspects

Tobias Rho, Günter Kniesel, Malte Appeltauer
Dept. of Computer Science III
University of Bonn
Römerstr. 164, D-53117 Bonn
Germany
{rho,gk,appeltau}@cs.uni-bonn.de

ABSTRACT
In theory, join points can be arbitrary places in the structure or execution of a program. However, most existing aspect languages do not support the full expressive power of this concept, limiting their pointcut languages to a subset of the theoretically possible join points. In this paper we explore a minimal language design based on only three built-in *fine-grained pointcuts*, which enable expressing the entire spectrum of structures of an underlying base language, from types to statements and expressions. The combination of fine-grained pointcuts with *uniform genericity* in our *LogicAJ 2* language yields the concept of *fine-grained generic aspects*. We demonstrate their power by showing how they allow programmers to express and extend the static primitive pointcuts of AspectJ and how they can model applications that previously required run-time reflection or special purpose language extensions.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features - *abstract data types*

General Terms
Languages

Keywords
Fine-Grained Genericity, Homogeneous Generic Aspect Language, Program Transformation, Multi Join Points, Fine-Grained Pointcuts

1. INTRODUCTION
The notion of join points is central to aspect-oriented programming languages. Join points are well-defined places in the structure or execution flow of a program [4], [5], [3], [21], [16]. In theory, they could be arbitrary program elements or run-time events. In practice, however, the classes of join points supported by most existing aspect languages are limited. Method call, method execution, field access and field modification are the typical join points that are widely supported. Different researchers [7], [6], [17], [9] have noted independently that finer grained join points are necessary in various application areas. For instance, Kniesel and Austermann [9] show that thorough code coverage analysis requires access to every individual statement in a program. They present a professional code coverage tool for Java based on load-time byte code adaptation [11]. Sullivan and H. Rajan [17] address the same problem domain but provide a solution at a higher level of abstraction. They show how code coverage analysis can be implemented in a language that provides statement-level join points. Their approach uses reflection and generates new aspects based on reflective information at join points. Unfortunately, this makes static type checking impossible.

Another impressive application area for fine-grained pointcuts is the automatic detection and optimization of highly parallel loops. Figure 1 shows a simple example. B. Harbulot and R. Gurd [7] demonstrate that with *AspectJ* [8] parallelization of such loops is only possible after refactoring the loop into a new method with a defined signature pattern that makes its lower and upper bounds explicit. Because typical code in parallel scientific applications almost never makes loop bounds explicit as methods the authors conclude that statement-level join points are needed to enable aspect-based optimizations of highly parallel programs. In *LoopsAJ* [6] they provide a solution tailored specifically to the interception of loops. However, their solution also imposes some constraints on the structure of the code in and before loops.

```
1 public void m(){
2   int[] a = new int[42];
3   int[] b = new int[42];
4
5   for(int i = 0;i< a.length;i++) {
6     a[i] = 2*i;
7     b[i] = i*i;
8   }
9 }
```

Figure 1. Simple highly parallel *for* loop

The work reported in this paper goes beyond previous approaches in that it provides a comprehensive design for fine-grained pointcuts in an extensible, statically checkable, high-level language. Although our aspect language, *LogicAJ 2*[1], provides only 3 built-in pointcuts, it is able to express all possible join points whose shadows are elements of the base language (in our case Java). We demonstrate the expressiveness of our concept by showing how it allows programmers to express and extend the static primitive pointcuts of AspectJ and how to model applications that previously required special purpose language extensions. In particular, we show that LogicAJ 2‘s combination of fine-grained genericity and extensibility can express the functionality of LoopsAJ, without imposing any constraints on the structure of base programs.

[1] LogicAJ 2 is based on the generic aspect language LogicAJ [10].

Section 2 introduces our language design. Section 3 demonstrates how the language can be easily extended by expressing 'basic' poincuts of AspectJ using fine-grained genericity. In Section 4 we give two examples of modelling applications that previously required specific language extensions: the Law Of Demeter and the for-loop pointcut. Section 5 discusses related work and Section 6 ongoing work. Finally, section 7 concludes.

2. FINE-GRAINED GENERICITY

In this section we gradually introduce the basic concepts behind LogicAJ 2 and illustrate them on small examples.

2.1 Basic Pointcuts

The aim of our design was the identification of a minimal, orthogonal set of pointcuts that are able to express more complex ones offered in other languages. Our analysis resulted in just *three basic pointcuts*, representing the distinct classes of basic elements of any programming language: declarations, statements and expressions. Their syntax is:

- `decl(join_point, declaration_code_pattern)`
- `stmt(join_point, statement_code_pattern)`
- `expr(join_point, expression_code_pattern)`

The first argument of each basic pointcut is an explicit representation of the matched join point (see Section 2.4). The second argument is a pattern describing the join point (see Section 2.2).

The expr pointcut selects any expression matching the given source code pattern and binds the join_point argument to an explicit representation of the matched join point. The stmt pointcut does the same for statements. These two pointcuts can match any element within a method body. The decl pointcut additionally matches declarations of classes, interfaces, methods and fields.

Taken together, these three pointcuts can match any structure of a base (Java) program, from the coarsest to the finest granularity. Therefore we also call them *fine-grained pointcuts*.

The basic pointcuts can be used to bind any syntax element of their domain, by omitting the code pattern.

2.2 Logic Meta-Variables

Unlike most aspect languages, we do not provide a special syntax for the patterns used to specify join points. Instead, join point descriptions are simply base language code or patterns resulting from the ability to use placeholders for all base language elements that are not syntactic delimiters or keywords.

Instead of unnamed wildcards our placeholders are named *logic meta-variables* (LMV). Meta-variables are variables that can range over syntactic elements of the base language (e.g. Java). They are denoted syntactically by names starting with a question mark, e.g. "?methodBody".

Named meta-variables give us the ability (1) to express that different occurrences of the same placeholder must agree on the matched value and (2) to use the matched values as a building block of advice code. Rho and Kniesel [10], [12] show that uniform use of logic meta-variables in pointcuts and advice (*uniform genericity*) increases the expressiveness, reusability and modularity of aspects – even without the added power of fine-grained pointcuts introduced here. The combination of uniform genericity and fine-grained basic pointcuts is called *fine-grained genericity*. We demonstrate the increased expressiveness of fine-grained genericity in Section 3 and 4.

In addition to logic meta-variables that have a one-to-one correspondence to individual base language elements, *logic list meta-variables* (LLMV) can match an arbitrary number of elements, e.g. arbitrary many call arguments or method parameters. These variables are indicated syntactically by two leading question marks, e.g. "??parameterList". Their introduction is motivated by the fact that in truly generic application scenarios one often needs to say things like "match every constructor invocation" or "add a forwarding method for every method from type T" or "select all update expressions in a for-loop". In such cases it is neither possible to know the exact number of parameters of an invocation or a method, nor is it practical to specify a finite set of method argument lists. The language provides a set of built-in operations on LLMVs, e.g. concatenation and member check.

Unnamed logic meta-variables are indicated by an underscore (?_ and ??_). If a pointcut contain several unnamed meta-variables, they are all treated as distinct variables.

2.3 Named Pointcuts

Using the basic pointcuts, programmers can define arbitrary custom pointcuts. Custom pointcut definitions can be named and can have meta-variables as arguments. Unlike in AspectJ, for instance, custom pointcuts can be defined recursively. This is useful for expressing transitive relationships, for instance the subtype relation. The recursive definition of a generalized version of AspectJ's withincode pointcut is discussed in Section 2.5.

2.4 Explicit Join Points

Our language design leverages on the power of meta-variables by making the join point selected by a basic pointcut explicit as a meta-variable argument. This is an extremely powerful concept, since it makes join points first class entities of the aspect language.

Figure 2 shows a pattern that selects if statements. Upon every match the ?if meta-variable is bound to the complete matched statement (the join point), whereas the meta-variables contained in the pattern are bound to the respective sub-elements of the statement. In this case ?cond is bound to the condition expression and ??someStatements is bound to the list of statements in the body of the if statements' block.

```
stmt(?if, if(?cond){??someStatements})
```

Figure 2. Selection of an if statement, its condition and its body

Figure 3 shows the use of two expr pointcuts that select all calls of the methods *foo* and *bar*. The matched join points are explicitly represented by the meta-variable ?jp, which is passed as a parameter to the pointcut definition. Thus, it can be used as the join point of an advice based on fooBarCalls(?jp).

```
1  pointcut fooBarCalls(?jp):
2      expr(?jp , foo() )
3   || expr(?jp , bar() )
```

Figure 3. Join points made explicit via meta-variables

Alternatively, we can reuse this pointcut as shown in Figure 4. Note that the meta-variable ?call is used within the if statement pattern in the stmt pointcut *and* within fooBarCalls. This way we express that the calls to foo and bar must be the condition of an if statement. Note further that the defined pointcut provides ?if and ?call as parameters to its users. Thus it does not predetermine whether the matched if statement or the matched call is the join point that it selects.

```
1 pointcut fooBarCallsWithinIf(?if, ?call):
2     stmt(?if, if(?call){??someStatements} )
3  && fooBarCalls(?call) ;
```

Figure 4. Refining the pointcut from Figure 3 to select calls of foo or bar that occur within an if-condition

Since different meta-variables can represent different join points at the same time it is possible to express relations between join points, as illustrated in Figure 4. This is an extremely powerful concept. In Section 4.1 we demonstrate how it enables a concise implementation of the Law of Demeter.

In addition it gives us the option to let a generic advice choose at which of the multiple join points the advice code should be woven. Therefore the syntax of LogicAJ 2 advice[2] was slightly extended. The target join point of the advice must be specified as the *first* argument of the advice. The advice shown in Figure 5 counts all invocations of foo or bar that occur as the condition of an if statement. If we change the advice parameter to ?if the advice will count the number of if statements whose conditions are calls to foo or bar.

```
1 around(?call):  fooBarCallsWithinIf(?if, ?call) {
2   Counter.count++;
3 }
```

Figure 5. Explicit choice of the effective join point for an advice

2.5 Meta-Variable Attributes

For many uses, it is not sufficient to consider only a syntactic element itself but also the static context of the element. For example, the declaring type is important information about a method or a field declaration. Similarly, the statically resolved binding between a method call and its called method or between a variable access and the declared variable is necessary for several pointcuts.

We make this information available via LMV *attributes*. An attribute *a* of a meta-variable ?*mv* is accessible via:

?mv::a

Figure 6 describes the attributes used in the remainder of the paper. They are a subset of the attributes supported by LogicAJ 2.

Attrib.	Represented context information of a LMV
parent	The enclosing element of the syntax element represented by the LMV.
ref	The statically resolved declaration referenced by an expression: a call references a method, and an identifier a field, variable or parameter declaration.
type	The statically resolved Java type of an element bound to a LMV. This attribute is syntactic sugar. It is inferable via the ref attribute.

Figure 6. LMV attributes provide additional information about the syntactic elements' context and the resolved Java bindings.

Figure 7 demonstrates the use of the parent attribute for the withincode pointcut, known from AspectJ. It checks if a join point is defined in the body of a given method. We present a generalized version that checks the withincode relationship of statements and expressions to any enclosing element.

```
1 pointcut withincode(?jp,?enclosing):
2     ( expr(?jp)  ||  stmt(?jp) )
3   && ( equals(?jp::parent, ?enclosing)
4        || withincode(?jp::parent, ?enclosing)
5      );
```

Figure 7. Definition of the `withincode` pointcut in LogicAJ 2

First, the ?jp variable is bound to an expression or statement. The *equals* predicate has a double role. If ?enclosing was bound to a value before withincode was called, *equals* just checks if the value of ?enclosing is ?jp's parent element Otherwise it binds ?enclosing to ?jp::parent. In order to get all the directly and indirectly enclosing elements, of ?jp the pointcut is evaluated recursively for the parent of the ?jp.

3. EXTENSIBILITY OF THE POINTCUT LANGUAGE

This section shows how basic pointcuts can be used to build pointcuts known from common aspect languages. Designing semantic meta-levels with basic pointcuts drastically enriches the usability of pointcuts and is an important criterion for the expressiveness of an AO language.

3.1 Static AspectJ Pointcuts

The pointcuts offered by AspectJ are very useful. Due to the expressiveness of our minimalist pointcut language we can define custom pointcuts that implement AspectJ pointcut semantics with little effort. We have already shown the implementation of the withincode pointcut in Figure 7. In this section we show the implementation of the call and get pointcut in LogicAJ 2. The other static pointcuts of AspectJ can be implemented following the same scheme.

3.1.1 Call Pointcut

Our implementation of the AspectJ call pointcut (Figure 8) starts with an expr pointcut selecting the call expression, (line 4) and a decl pointcut binding the called method (lines 5-6). The equals predicate in line 7 denotes that the call join points are statically bound to the method ?method. Line 8 binds the declaring type of ?method to ?declType by using the ref and type attributes (see Line 8). Line 9 binds the ??parTypes list to the types of the method parameters. We omit the definition of the parameterTypes pointcut. It can easily be implemented as a recursive pointcut using the type argument.

```
1 pointcut call(?jp, ?declType, ??modifiers,
2               ?returnType, ?name, ??parTypes):
3
4     expr(?jp, ?name(??args) )
5  && decl(?method,
6       ??modifiers ?returnType ?name(??par){??stmts} )
7  && equals(?method, ?jp::ref)
8  && equals(?declType, ?method::parent::type)
9  && parameterTypes(??parTypes,??par);
```

Figure 8. Implementation of the call pointcut

Figure 9 shows two usage examples of the call pointcut. For comparison, the respective AspectJ syntax is shown as a comment. Line 1-3 illustrates the selection of calls to the method with signature m(int) in class Foo. Line 5-7

```
1 //AJ:  pointcut m(): call( void Foo.m(int) )
```

[2] and declare error/warning constructs

```
2 pointcut m(?jp):
3      call(?jp, Foo, ??_, void, m, [int]);
4
5 //AJ:  pointcut anycall(): call(* *.*(..));
6 pointcut anycall(?jp): call(?jp,?_,??_,?_,?_,??_);
```

Figure 9. Comparison between AspectJ and LogicAJ 2 call syntax. Square brackets (line 3) denote a list of values.

3.1.2 *Get Pointcut*

Our next example implements AspectJ's `get` pointcut. It selects field read accesses. Field declarations can have different syntactic forms. For instance, they can be declared with or without a value assignment. Each syntactic occurrence selects a different set of join points. Figure 10 presents the unification of the different syntactic join point variants by a more general field access pointcut. The union is expressed by the disjunction in lines 4-5, which states that all declarations with *or* without value assignment are bound to ?field.

```
1 pointcut get(?jp,??modifier,?declType,?name,?retType):
2      expr(?jp,?name )
3   && (
4           decl(?field,??modifier ?retType ?name; )
5        || decl(?field,??modifier ?retType ?name = ?v;)
6      )
7   && equals(?field, ?jp::ref)
8   && equals(?declType, ?field::parent::type);
```

Figure 10. Implementation of AspectJ get pointcut semantics

In this example, join points are selected on the syntactic, not on the semantic level. However, we do not see this as a limitation of our approach. The defined custom pointcut implements a semantic selection criterion. It can be reused, hiding the syntactic details.

3.2 New Pointcuts

In the following, we give examples illustrating how easy it is to define additional semantic pointcuts that are neither built-in nor expressible in common AO languages.

3.2.1 *Local Variable Access Pointcuts*

As a complement to the common get and set pointcuts that select fields, we introduce getL and setL pointcuts that select read and write accesses to *local* variables. We describe the implementation of the getL pointcut in detail. The setL implementation is analoguous.

```
1 pointcut getL(?jp,?type,?name):
2     //Select all identifier expressions:
3     expr(?jp, ?name)
4     // Select all local variable declarations:
5     &&(   stmt(?localdecl, ?type ?name;)
6        || stmt(?localdecl, ?type ?name = ?val;)
7        )
8     // Check that the local variable declaration
9     // is referenced by the identifier:
10  && equals(?localdecl, ?jp::ref) );
```

Figure 11. Implemantation of new pointcuts: the getL pointcut only selects local variable accesses

The intended semantics of getL is to select all identifiers whose declaration is a local variable. We start by selecting all expressions that have the form of an identifier, that is, consist of a single name. This is done by the expr(?jp, ?name) pointcut (Figure 11, line 3). The set of identifiers matched this way can also contain fields and parameters. In order to understand how we limit it to local variables only, it is helpful to recall that, in Java, the declaration of a local variable is a *statement*. Accordingly, we enclose each of the code patterns corresponding to a variable declaration into a stmt pointcut (see Figure 11, lines 5-6). The final equals predicate checks whether the selected local declaration is indeed the declaration of the identifier matched in line 3.

3.2.2 *Field Pointcut*

Within the declarations of the get pointcut we had to consider syntactic differences of field declarations. We can encapsulate those within a field pointcut definition in order to achieve a more modular and readable implementation of get and other pointcuts that deal with field declarations. We will see another use of the field pointcut in Section 4.1.

The definition illustrated in **Figure 12** selects all field declarations, with and without initializers. Then it determines the declaring type by accessing the static join point context captured by the meta-variable attributes ?jp::parent::type. Similarly, we can implement a method or a class pointcut that abstracts from the syntactic variants of the base language.

```
1 pointcut field(?jp,?declaringType,?returnType,?name):
2   (
3        decl(?jp, ?returnType ?name; )
4     || decl(?jp, ?returnType ?name = ?anyVal;)
5   )
6   && equals(?declaringType, ?jp::parent::type);
```

Figure 12. `field` semantics

4. EXAMPLES

We now consider two different use cases that rely on fine-grained pointcuts. Section 4.1 presents a simple check for the static variant of the Law of Demeter, expressing a contract previously claimed to require extension of *AspectJ* by *statically executable advice* [14]. Section 4.2 comes back to the high-performance computing example.

4.1 Example: Law Of Demeter

Binding several join points at the same time enables very expressive pointcuts. This section gives a thorough example. It shows how the declare warning construct can be used to check the Law of Demeter with the help of a fine-grained pointcuts.

The *Law of Demeter* (*LoD*) [13] restricts the method calls used in a class *C* to methods from 'known' types. These include

1. *C*,
2. the types of the calling method's parameters,
3. the types of *C*'s fields,
4. the return types of *C*'s methods
5. the classes instantiated in *C and*
6. all the supertypes of any of the known types from 1-5.

Figure 13 shows the *LawOfDemeter* aspect, concisely implemented in *LogicAJ 2*. The aspect uses the custom pointcuts method, constructorcall and the subtype, which can be defined easily, like the pointcuts in Section 3.

The pointcut knownTo defined in line 3-11 encapsulates rules 1-5 of the LoD. It defines the basic set of types known to a method. Its semantics is that ?Type is known to a method with parameter types ?ParameterTypes contained in ?CallingType.

- Line 4 checks the first rule: The ?CallingType is known to itself.

- Line 5 checks the second rule: Every member of ?ParamTypes is known. The member predicate successively binds ?Type to an element of ?ParamTypes.
- Line 6 checks the third rule: The type of every field of ?CallingType is known. The field predicate successively binds ?Type to the type of a field of ?CallingType.
- Line 7 checks the fourth rule: The return type of every method of ?CallingType is known. The method predicate successively binds ?Type to the return type of a method of ?CallingType.
- Lines 10-11 check the fourth rule: The type instantiated by a constructor call contained in the ?CallingType is known. The constructorcall predicate in line 9 determines all constructor calls and the withincode predicate in line 11 ensures that only calls within ?CallingType are regarded.

The knownTo pointcut is the core of the real LoD checking in line 13-25. Lines 14-17 set the stage by binding the meta-variable ?CalledType to the static receiver type of a method call, ?CallingType and ?CallingMethod to the type and method containing the call, and ??ParamTypes to the list of parameter types of ?CallingMethod. Line 21 uses the knownTo pointcut to determine a known ?Type and line 22 verifies whether ?CalledType is a supertype of the known ?Type[3]. If not, a violation is reported in line 24 along with the violating call.

This example uses explicit join point LMVs (see Section 2.4). The call and the constructorcall pointcuts are used within a single scope: the one in line 14 and the nested one in the definition of knownTo (line 9). This example could not be expressed if the join points could not be explicit parameters of the pointcuts in line 9 and 10. If there were only implicit join points (as in *AspectJ*), it would not be clear that constructor calls should not be reported as LoD violations and that failure of matching constructor calls in line 9 should not inhibit reporting a violation of other calls.

```
1 aspect LawOfDemeter {
2
3  pointcut knownTo(?CallingType,??ParamTypes,?Type):
4      equals(?Type, ?CallingType)                 //rule 1
5   || member(?Type, ??ParamTypes)                 //rule 2
6   || field(?jp,?CallingType, ?Type, ?Fname)      //rule 3
7   || method(?jp,??_,?Type,?CallingType,?Mname,??_)//r.4
8   ||
9   (
10        constructorcall(?ConstrCall,?Type,??_)  //rule 5
11     && withincode(?ConstrCall, ?CallingType )  //rule 5
12   );
13
14  declare warning:
15      call(?called,?CalledType,??_,?_,?CalledMeth,??_)
16   && method(?call,??_,?_,?CallingType,?CallingMeth,
17               ??ParamTypes )
18    && withincode(?called, ?call)
19    &&
20    !(
21           knownTo(?CallingType,??ParTypes,?Type)//r.1-5
22        && subtype(?Type, ?CalledType)          //rule 6
23      )
24     : "The call violates the Law of Demeter.";
25 }
```

Figure 13. Aspect checking the Law of Demeter (LoD) at weave-time. It reports a warning for every method invoked on a type that is not among those "known" to the calling type or the parameters of the calling method.

[3] Note that every type is a super type of itself. Subtype can be declared recursively, similar to withincode (see Figure 6)

4.2 Example High-Performance Computing

The following section shows how the execution of highly parallel loops can be distributed by a generic aspect onto a set of threads, following the approach described in [6]. Unlike the approach in [6] our solution does not rely on code conventions.

The target of a high-performance aspect could be the following for-loop, whose body uses the local array variables a and b.

```
1 public void m(){
1     int[] a = new int[42];
2     int[] b = new int[42];
3
4   for(int i = 0;i< a.length;i++) {
5     a[i] = i*i;
6     b[i] = i*i*i;
7   }
8 }
```

Figure 14. Simple `for`-loop

The detection of the *highly parallel* loop can be performed with the simple detector presented in Figure 15. The pointcut highlyParallelLoop checks that no method call is present in the block (avoiding side-effects) and tests that values are exclusively read from (or written to) a variable in the block.

This ensures that the order of the computation does not affect the result. Below we see the pointcut describing these checks. We assume to have implemented withincode, set, setL, get and getL pointcuts as shown in Section 3.

```
1 pointcut highlyParallelLoop(?jp,?range,?lb,
2                             ?ub,?body,?incr):
3     //selects for-loops
4  stmt(?jp,
5     for(?type ?range=?lb; ?range<?ub; ?incr){?body})
6     // selects all join points within the loop body
7  && withincode(?stmts, ?body)
8     // excludes calls within loops
9  && !call(?stmts,?_,??_,?retType,?name,??args)
10 && // no read AND write access to a variable allowed
11 !(
12       setL(?bodyJPs, ?body, ?type, ?name, ?val)
13    && withincode(?another, ?body)
14    && getL(?another, ?body, ?type, ?name, ?val)
15 )
16 &&  // the same for fields
17 !(
18       set(?bodyJPs,?p,?cl,?meth,??mod,?type,?name,?val)
19    && withincode(?another, ?body)
20    && get(?another,?p,?cl,?meth,??mod,?type,?name,?val)
21 );
```

Figure 15. The pointcut selecting highly parallel loops.

The around advice in Figure 16 wraps the for-loop into a run() method of a thread and modifies the bounds of the loop. The for loop now runs in parallel in several threads, each thread calculating only a uniform range of the loop.

```
1 around(?jp) :
2   highlyParallelLoop(?loop,?range,?lb,?ub,?body,?incr){
3
4    int THREADS = 5;
5    //resolve the value of the bounds
6    final ?ub::type ub = ?ub;
7    final ?lb::type lb = ?lb;
8    final ?range::type range = (ub - lb)/THREADS;
9    List list = new ArrayList();
10
```

```
11   for(int threads=0; threads < THREADS; threads++){
12     final int finalThreads = threads;
13     Thread thread = new Thread() {
14       public void run(){
15         ?range::type newLb = lb+range*finalThreads;
16         ?range::type newUb = newLb + range;
17
18         if(newUb >= ub)
19           newUb = ub;
20
21         for(?range::type ?range = newLb;?range < newUb;
22            ?incr){
23               ?body
24         }
25       }
26       thread.run();
27       list.add(thread);
28     }
29
30    for(int threads = 0;threads<list.size();threads++)
31       try{
32           ((Thread)list.get(threads)).join();
33       }
34        catch(InterruptedException e){
35           e.printStackTrace();
36        }
37     }
38   }
```

Figure 16. The for-loop parallelization aspect.

5. RELATED WORK

A comparison with existing generic aspect languages is given in [12]. It includes a thorough comparison of LogicAJ 2 to related work from logic meta-programming and program transformation systems. Here we confine ourselves to related work specific to fine-grained genericity.

The *TyRuBa* language [19] introduces logic meta-programming for Java programs by defining Prolog-like predicates on Java programs. All code blocks of the base program are represented as quoted Java code within the *TyRuBa* rules. The quoted code may contain meta-variables for types and identifiers. In *TyRuBa* the quoted code can only be used for the generation of Java code, but not in the query language.

Several approaches exist that describe finer-grained extensions to the *AspectJ* join point model.

The extensible AspectJ compiler *abc* [1] provides a Java API to extend the set of known *AspectJ* pointcuts. For every additional pointcut, the lexer, parser and weaver must be extended to support the new pointcut. Implementation of custom pointcuts in LogicAJ 2 does not require any such changes. Es we have shown, the extensions are expressible within the language.

EOS-T [17] extends the *AspectJ* primitive pointcuts with pointcuts for conditionals and loops. It does not provide no ability to refer to join points statement-arguments or -blocks. Harbulot presents *LoopAJ* [6], an AspectJ extension for loop pointcuts. His approach is bult on abc and uses byte code analysis to identify loops. Kniesel and Austermann [9] present a professional code coverage tool for Java, *CC4J*, implemented based on the *JMangler* load-time adaptation framework [11]. Working at byte code level, however, is not the preferred level of abstraction for most programmers.

Borba et al. introduce *JaTS* [2], a language for pattern based transformations of Java programs. Similar to our basic pointcuts, code patterns are used to describe program parts on which transformations should take place. The transformation specification is described with another pattern. Like in *LogicAJ* 2, both parts can be linked by the use of meta-variables, which substitute syntactic elements at the interface level of a base-program. According to personal communication with the authors, meta-variables can also match finer grained elements. That lets JaTS appear to be the closest match for our concept of fine-grained genericity. Comparison of *JaTS* and *LogicAJ* 2 will therefore be a rewarding topic of future work.

6. ONGOING WORK

The design of LogicAJ 2 described in this paper is currently implemented as an extension of our existing LogicAJ compiler, which is available at [15].

The added expressive power of a generic aspect language does not come for free. In particular, static analysis of aspect code is difficult in the presence of meta-variables.

In order to prevent substitution of statements where expressions are expected and vice-versa, meta-variables need to be *syntactically typed*, that is every meta-variable needs to have a type that determines the kind of syntactic entity from the base language that may be substituted. Syntactic types can either be declared or inferred from the definition of the predicates that are used to bind meta-variable values. For lack of space, we did not address this issue in this paper. This is a topic of ongoing work.

Currently we do not support dynamic join points like cflow, this or target with our basic pointcut model. In contrast to static join points, dynamic ones have no counterpart in the base program that could be described by a unique code pattern. Overcoming this limitation is also subject of ongoing work.

We will evaluate LogicAJ 2 by applying fine-grained genericity to different application areas. General software transformation approaches, like [18] have addressed optimizations techniques like partial evaluation and data-flow optimization with generic transformations. We will analyze how they can be translated to fine-grained generic aspects.

Contract4J [20] uses *AspectJ* to check contracts on Java. Currently the contracts are limited to *AspectJ* join points. For instance loop invariants can not be checked. Fine-grained genericity could be used to remove this restriction.

7. CONCLUSION

In this paper, we have introduced the concept of fine-grained genericity for aspect languages. Our approach is based on a minimal set of fine-grained pointcuts and base-language code patterns containing logic-meta variables. This enables us to express context-dependent aspect effects and dependencies between multiple join points. In addition, we have shown that fine-grained genericity is able to express the static pointcuts known from *AspectJ* and to define arbitrary other kinds of pointcuts that previously required specific language extensions.

Thus, we have shown that there is no need for extending an aspect language in order to implement new ‘basic’ pointcuts if the language itself is powerful enough to select *all* base-language join points.

8. ACKNOWLEDGEMENTS

We want to thank the FOAL’06 reviewers for their constructive and knowledgeable comments on the submitted draft of our paper.

9. REFERENCES

[1] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. abc: An extensible AspectJ compiler, Transactions on Aspect-Oriented Software Development, 2005

[2] Castor, F., and Borba, P. A language for specifying Java transformations. In V Brazilian Symposium on Programming Languages, pages 236-251, Curitiba, Brazil, 23rd-25th May 2001.

[3] Conejero, J. M., van den Berg, K., and Chitchyan, R. Aosd ontology. http://www.aosdeurope.net.

[4] Filman, R. E., and Friedman, P. D. Aspect-Oriented Programming is Quantification and Obliviousness, Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000, Minneapolis.

[5] Filman, R. E., Elrad, T., Clarke, S., and Aksit, M. Aspect-Oriented Software Development. Addison-Wesley, Boston, 2005.

[6] Harbulot, B., and Gurd, J. R. A join point for loops in aspectj. FOAL 2005, 2005.

[7] Harbulot, B., and Gurd, J. R. Using AspectJ to Separate Concerns in Parallel Scientific Java Code, Proceedings of 3rd International Conference on Aspect-Oriented Software Development (AOSD), March 2004

[8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. An Overview of AspectJ, Proceedings of ECOOP 2001

[9] Kniesel, G., and Austermann, M. CC4J - Code Coverage for Java, Lecture Notes in Computer Science, Volume 2370, Jan 2002, Page 155

[10] Kniesel, G., and Rho, T. Beyond Type Genericity - Homogeneous Genericity for Aspect Languages, Technical Report IAI-TR-2004-4, University of Bonn, Dec 2004.

[11] Kniesel, G., Costanza, P., and Austermann, M. JMangler - a framework for load-time transformation of Java class files Source Code Analysis and Manipulation. Proceedings. First IEEE International Workshop on 10 Nov. 2001 Page(s):98 – 108

[12] Kniesel, G., Rho, T., Generic Aspect Languages - Needs, Options and Challenges, Special issue of L'Objet, Hermes Science Publishing, London, 2006

[13] Lieberherr, K. J., Holland, I., Riel, A.: Object-Oriented Programming: An Objective Sense of Style, *In Proceedings of the Conference on Object-oriented Systems, Languages and Applications (OOPSLA)*, San Diego, California, USA, pages 38-48, September 1988.

[14] Lieberherr, K., Lorenz, D., and Wu, P. A case for statically executable advice – checking the law of Demeter with AspectJ, Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, MA, March 17-21, ACM Press, 2003

[15] LogicAJ Homepage, http://roots.iai.uni-bonn.de/research/logicaj

[16] Masuhara, H., Kiczales, G., and Dutchyn, C. A Compilation and Optimization Model for Aspect-oriented Programs. Compiler Construction (CC) also Lecture Notes in Computer Science (LNCS) vol. 2622.

[17] Rajan, H., and Sullivan, K. "Generalizing AOP for Aspect-Oriented Testing", In the proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD 2005), 14-18 March, 2005, Chicago, IL, USA

[18] Visser, E., Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In C. Lengauer et al., editors, Domain-Specific Program Generation, *volume 3016 of Lecture Notes in Computer Science*, pages 216--238. Spinger-Verlag, June 2004.

[19] Volder, K. D. Aspect-oriented logic meta programming. In Proceedings of the Second International Conference on Metalevel Architectures and Reflection, volume 1616 of Lecture Notes in Computer Science . Springer-Verlag,1999

[20] Wampler, D., Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces, Fifth AOSD Workshop on ACP4IS, Bonn 2006

[21] Wand, M., Kiczales G., et al. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming, LNCS 2196, 2001