

Relational Aspects for Context Passing Beyond Stack Inspection

Rémi Douence

Obasco group EMN-Inria, Lina

Ecole des Mines de Nantes

4 rue Alfred Kastler, La Chantrerie, BP 20722, Nantes Cedex 3
France

<http://www.emn.fr/douence>

ABSTRACT

In Java, context passing can be implemented by adding an argument to every method so that a value (i.e., context) is passed down the call stack to methods that require it. In AspectJ, context passing can be implemented by simply defining a pointcut with the `cflow` operator. In this article, we identify the essence of context passing in AspectJ, and generalize it beyond stack inspection. Our approach relies conceptually on databases and logic queries. When a method is called in the base program, it creates a relation between its receiver and its arguments. We use a logic language, Datalog, as a pointcut language in order to compose relations and pass context. This language has nice properties such as: queries always terminate, its fix point based semantics is declarative, recursion makes it expressive. A couple of Java examples, including a constraint solver and the JHotDraw GUI framework, illustrate the benefits of our relational aspects.

1. OVERVIEW

Context passing is a natural application of AOP. Indeed, it requires to specify (at least) two points of interest. At one point context is collected. At the other point the collected context is accessed. In other words: context passing cross-cuts the base program. For instance, AOP has been used to pass file opening mode to the prefetch module in an operating system [7], and to select top-level figures (by passing hierarchical context) to the refresh method in a graphical editor [2].

AspectJ [21] provides a pointcut language to define join-points (i.e., execution points) where an advice is to be executed. Basically, a pointcut specifies a method signature, but AspectJ also provides a pointcut construction `cflow` devoted to context passing. For instance, let us consider the following aspect `Bill`.

```
1 aspect Bill {
2   before(Caller c, Worker w):
3     execution(void Worker.task()) && target(w)
4     && (cflow(execution(void Caller.service())
5             && target(c)) {
6       w.bill(c);
7     }
8 }
```

It can be read as "when a worker performs a task (line 3) *during* the execution of `c.service` (line 4-5), the worker must bill its caller `c` (line 6)". There is context passing because although there can be an arbitrary number of intermediate nested calls, the value of `c` is available when `task` is executed. Such a context passing pointcut can easily be simulated with non context passing pointcuts as follows:

```
1 aspect BillWithoutCflow {
2   Stack callers = new Stack();
3   before(Caller c):
4     execution(void Caller.service())
5     && target(c) {
6     callers.push(c);
7   }
8   after():
9     execution(void Caller.service()) {
10    callers.pop();
11  }
12  before(Worker w):
13    execution(void Worker.task()) && target(w) {
14      if (!callers.isEmpty()) {
15        w.bill(callers.peek());
16      }
17    }
18 }
```

The first advice, line 6, collects the caller identity at the beginning of `service` execution. The second advice, line 10, discards the caller identity at the end of the execution of `service`. Finally, the third advice, lines 14-15, checks whether an execution of `task` occurs within an execution of `service` and if so retrieves the caller identity and bills it. So, in AspectJ, context passing is (conceptually) based on call stack inspection. This is quite limited, because, in general, context is not necessarily in the call stack.

For instance, let us now consider a slightly different scenario: in a batch sequential application `service` does not trigger a task immediately but it returns a `RequestId`. Later, this

`RequestId` is passed as a parameter to `task`. In this case, `service` has already returned when `task` is called, so it is not possible anymore to relate a `Caller` and a `Worker` with a control flow based pointcut. Two method executions are related because they share a reference: the return value of a method is the argument of the other. Such a dependency chain crosscuts the execution trace (a.k.a. history) of the base program rather than its call stack. The following aspect collects context (i.e., pairs `Caller` and `RequestId`) in a hashtable (line 6). When a task is performed, the corresponding caller is accessed in the context (line 12).

```

1 aspect BillBatch {
2     Hashtable callers = new Hashtable();
3     after(Caller c) returning(RequestId rid):
4         execution(RequestId Caller.service())
5         && target(c) {
6         callers.push(rid,c);
7     }
8     before(Worker w,RequestId rid):
9         execution(void Worker.task(RequestId))
10    && target(w)
11    && args(rid) {
12        w.bill(callers.get(rid));
13    }
14    after():
15    execution(void Worker.task(RequestId))
16    && args(rid) {
17        callers.remove(rid);
18    }
19 }

```

Here, the chain of dependency has only two links. In this paper, we propose relational aspects for generalizing context passing along such dependency chains of arbitrary length. Context (pairs, triplets,...) is collected in a `Set` instead of a `Stack` because we are interested in dependencies but not in ordering between context elements. Moreover, a piece of collected context can survive the method execution that collected it. We use Datalog [6] to define chains. Indeed, there is no need to design a new pointcut language when Datalog is a simple declarative language that is ideal to express dependency chains of arbitrary length. A relational aspect is an AspectJ aspect whose advices create Datalog facts (i.e., collect context), remove Datalog facts (i.e., discard context), or query Datalog facts (i.e. access context) when a method is called or returns in the base program. Relational aspects are Datalog embedded in AspectJ.

In this article, we argue relational aspects (and Datalog as a pointcut language) have many advantages. In particular:

- They are non invasive. Relational aspects pass context, so they do not modify the semantics of the base program (although advices that use the context can). Context passing cannot introduce non termination.
- They are declarative. Relational aspects collect (and deduce) relationships between objects independently of the base program execution order (i.e., tuples creation order).
- They are expressive as exemplified by the different applications in the next section.
- They are portable. Our proposal can be applied directly to other languages than Java.

```

1 aspect Inverse {
2     after(Set s,Elt e):
3     execution(void Set.add(Elt))
4     && target(s) && args(e) {
5         BELONG(e,s);
6     }
7     around(Elt e):
8     execution(void Elt.removeFromAll())
9     && target(e) {
10        for s in BELONG(e,s) do {
11            s.remove(e);
12        }
13    }
14    after(Set s, Elt e):
15    execution(void Set.remove(Elt))
16    && target(s) && args(e) {
17        ! BELONG(e,s);
18    }
19 }

```

Figure 1: Inverse Relation as a Relational Aspect

The rest of this paper is structured as follows. In Section 2, we illustrate how our relational aspects based on Datalog can be used for context passing in various applications, ranging from a toy program to the JHotDraw framework. Then, we review related work in Section 3. Finally, we evaluate the advantages of our approach and discuss future work.

2. RELATIONAL ASPECT APPLICATIONS

In this section, we define relational aspects in order to exemplify advantages of Datalog as a pointcut language for context passing. The considered base programs range from simple data structures to a complex graphical framework. We introduce Datalog notions as needed (see [6] for a detailed presentation).

2.1 Inverse Relation

Let us consider a class `Set`. It provides the methods `add` to add an element, `remove` to remove an element and `iterator` to enumerate its elements. However, when an element must be removed from all sets, no method enumerates the sets an element belongs to. This inverse relation can be defined with the relational aspect `Inverse` in Figure 1. Each time `add` is called, the first advice, line 5, adds a pair `(e,s)` to the relation `BELONG` (i.e., context is collected). A relation is a set of tuples, and a tuple aggregates Java values (references, integers...). Two tuples in the same relation are equals when they aggregate the same values. When an element `e` must be removed from all sets, the call to `e.removeFromAll()` (that applies `remove(e)` to *all* sets) is replaced by the second advice, lines 10-11, that queries `BELONG` (i.e., context is accessed) and applies `remove(e)` *only* to the sets that contain `e`. In this paper, we introduce a syntactic sugar `for` in order to simplify iterations. Finally, when `s.remove(e)` is called, the corresponding pair `(e,s)` is removed (i.e., context is discarded) by the third advice, line 17, to maintain the relation `BELONG`.

This first example highlights two advantages of our relational aspects. First, a relation has no orientation and can be enumerated in different ways (e.g., `for s in BELONG(e,s)` *versus* `for e in BELONG(e,s)`). Second, our relational aspects are abstract: the programmer does not have to precise how

```

1 aspect UpdateDisplay {
2   before(Display d):
3     execution(void Display.drawAll())
4     && target(d) {
5       ! ASSOC(d,--,.-);
6     }
7   after(Display d, FigureElement fe, Object o):
8     cflow(
9       cflow(execution(void Display.drawAll())
10            && target(d))
11            && execution(void FigureElement.Draw(Display))
12            && target(fe))
13    && get(* *.*.)
14    && target(o) {
15      String id = thisJoinPointStaticPart
16        .getSignature()
17        .getName();
18      ASSOC(d,fe,o,id);
19    }
20  after(Object o):
21    set(* *.*.)
22    && target(o) {
23      String id = thisJoinPointStaticPart
24        .getSignature()
25        .getName();
26      for all (d,fe) in ASSOC(d,fe,o,id) {
27        d.draw(fe);
28      }
29    }
30 }

```

Figure 2: Display Updating as a Relational Aspect

the relation `BELONG` is represented (e.g., by introducing a new field in `Elt` or by defining a hash table).

2.2 Expressive pointcuts for Increased Modularity

A recent article [25] proposes to use Prolog predicates to query a data base that contains static as well as dynamic information (in particular the full execution trace) in order to define pointcuts. Such expressive pointcuts increase modularity: they are more robust than syntactic ones and can remain valid when the base program is refactored. Their running example is a graphical application where figure element modifications should trigger display updating. They argue a pointcut should implement a semantics property such as: *“detect changes that occur on fields that were previously read in the control flow of the last `drawAll` call”*. Such a property can be detected by a relational aspect as in Figure 2.

The first advice, at line 5, deletes every fact about a display when its method `drawAll` is to be called. This way, collected facts represent only the most recent execution of `drawAll`. The second advice, at line 18, collects the reference `o` and the field `id` of every object accessed during an execution of `drawAll`. It also collect the reference `fe` of the figure element drawn. When the field of this object is modified later, the Datalog base is queried and the corresponding figure elements are updated in the display by the third advice at line 26 and 27.

Our relational aspect does not use the predicate `reachable` [25] that inspects the store in order to check if there is a path between two objects in the object graph, but it could be also

expressed in Datalog. However, it requires to collect every assignments which is not realistic. When the programmer knows (a subset of) the fields that can create such a path, a relational aspect could very easily maintain a partial object graph and define `reachable` as its transitive closure.

2.3 Explanations in a Constraint Solver

A constraint solver starts with variables, domains (i.e., possible values of variables), and constraints (i.e., compatible values for each pairs of variables). It first propagates constraints: it deletes from domains incompatible values according to constraints. These deletions are called removals. Second, it deletes all but one value for a variable (this is called a choice) and propagates constraints again. After several iterations either the constraints are satisfied and each variable has a single value in its domain, or one variable has an empty domain and there is no solution (this is called a failure).

We have shown previously [11] how explanations of removals [20] could be introduced with AspectJ in a constraint solver Cacao. Explanations in constraint solvers is a dynamic dependency analysis that enables partial backtrack. In case of failure, the solver does not backtrack to a previous state, but instead exploits dependencies in order to undo only the choices that are responsible for the current failure.

This analysis can be defined as a relational aspect (Figure 3) that can be decomposed in three parts. First, a data structure must be reified. In the solver we consider, a constraint is represented extensionally by a list of authorized pairs of values. The corresponding inverse constraint is not represented by the same list where each pair is flipped, but by a subclass whose methods flip the parameters when accessing the original list. The relations `SUPPORT` and `INVERSE` and the two first rules, lines 13-14 and 16-17, define `SUPPORTBOTHWAYS`, which enumerates all pairs of values *both ways*. Second, the choice constraints must be identified. A choice (i.e., a solver variable assignment) is encoded by a constraint of twice the same variable (see the rule for `CHOICE`, line 25, it has a non linear pattern with two occurrences of `var`). Third, the explanations are defined. A rule, lines 34-35, sows the seed of explanations when a value is removed because of a choice constraint. And the other rule, lines 37-40, propagates explanations: a value is removed because its supports have been removed. This recursive rule defines chains of explanations of arbitrary length.

We have defined explanations for constraint solvers as a simple relational aspect with only four context-collecting advices and three inferred relations (Figure 3). The corresponding AspectJ version [11] requires one hundred lines of code. Moreover, our relational version is easier to understand because it does not rely on data structures (e.g. introduction of fields in classes) or the order of the method calls which create tuples. So, it should be easy to reuse with different solvers.

2.4 Connected Figures in JHotDraw

JHotDraw is a Java GUI framework for technical and structured graphics. It has been developed as a “design exercise” but is already quite powerful. The framework comes with a drawing application that provides (among other things) fig-

```

1 aspect ExplanationsOfRemovals {
2   // 1) reify data structure
3   after (Relation r, Value val1, Value val2):
4     execution (Relation Relation.add (Value, Value))
5     && target (r) && args (val1, val2) {
6       SUPPORT (val1, r, val2);
7     }
8   after (Relation r) returning (Inverse r1):
9     call (Inverse.new (Relation))
10    && args (r) {
11      INVERSE (r, r1);
12    }
13   SUPPORT (val1, r, val2)
14     => SUPPORTBOTHWAYS (val1, r, val2)
15
16   SUPPORT (val1, r, val2) && INVERSE (r, r1)
17     => SUPPORTBOTHWAYS (val2, r1, val1)
18
19   // 2) identify choice relation
20   after (String s, Variable var1, Variable var2):
21     call (Relation.new (String, Variable, Variable))
22     && args (s, var1, var2) {
23       CONSTRAINT (r, var1, var2);
24     }
25   CONSTRAINT (r, var, var) => CHOICE (r)
26
27   // 3) explanations analysis
28   after (Relation r, Variable var, Value val):
29     cflow (target (r) && call (Removal revise ()))
30     && args (var, val)
31     && call (Removal.new (Variable, Value)) {
32       REMOVAL (var, val, r);
33     }
34   REMOVAL (var, val, r) && CHOICE (r)
35     => EXPLANATION (val, r)
36
37   REMOVAL (var, val, r)
38     && SUPPORTBOTHWAYS (val, r, val2)
39     && EXPLANATION (val2, r2)
40     => EXPLANATION (val, r2)
41 }

```

Figure 3: Explanations in a Solver as a Relational Aspect

ures (e.g. squares) and connections (i.e., lines between two figures which are maintained when a figure is moved). A figure can have several connections with different figures. Connections can form cycles. A connection end can be changed from one figure to another. The color of a figure or a whole (hierarchical) group of figures can also be edited.

We now define an aspect that detects connected figures so that when the color of a figure is edited, all connected, but only connected, figures also change colors (i.e., connection crosscuts groups). When two figures are connected in JHotDraw, three types of entities are involved: **Figure**, **Connection-Figure** (i.e., the connection), and **Connector**, an intermediary entity between a figure and a connection. Here, the problem is that a connector aggregates a figure, but a figure does not aggregate its connectors, so it is not possible to write code that inspects fields of instances and follows references to discover connected figures. (We miss an inverse relation as in the example of Section 2.1).

So we define a relational aspect (Figure 4). The five first advices collect relations between these entities. Two of these advices suppress tuples when a **Connector** is disconnected from a **ConnectionFigure**. Note that, we use `_` as a wild-

card notation at lines 8 and 16. The relation **CONNECTED** defines pairs of connected figures with only three rules. The first rule detects a direct connection between two figures: from a **Figure**, to a **Connector**, to a **ConnectionFigure**, to a **Connector**, to a **Figure**. The second rule states that the relation **CONNECTED** is transitive. The last rule states that this relation is commutative. Finally, when the color of a figure is changed, a query is evaluated to enumerate connected figures (line 33) whose color is updated accordingly (line 34).

JHotDraw is a well structured framework. However, it is quite generic hence complex. Repaint propagation could also be implemented by exploiting the infrastructure (e.g., listeners) of JHotDraw, but this requires a significant expertise in this framework. Moreover, when connected figures form a cycle, notifications should not introduce non termination. Our relational aspects cannot introduce non termination (Datalog queries always terminate). In order to evaluate our approach, we downloaded JHotDraw source code, browsed it and implemented this relational aspect. The full process was only one day long.

3. RELATED WORK

This article uses Datalog as a pointcut language for context passing. Our work is based on multiple sources (AOP, oo database systems, program analysis...).

AspectJ [21] provides a pointcut operator **cflow** to pass context downward between nested method calls. Our pointcut operator **path** [12] based on call tree, passes context upward from a subcallee to a caller. Both operators can be defined as a relational aspect. However, when the base program is recursive, control-flow information (e.g., call stack level) must be added to our tuples in order to distinguish nested recursive calls.

Alpha [25] is an AO extension of a toy OO core language implemented as an interpreter in Java. Pointcut definitions are full Prolog queries over a database of both static and dynamic information about the program or program execution. Prolog enables to raise the abstraction level of pointcut definitions which can remain valid in presence of refactoring of the base application. Prolog is more expressive than Datalog. Prolog is Turing Complete, Datalog is not. However, Prolog is less declarative than Datalog (Prolog semantics relies on the clause definitions order and the cut operator). Prolog queries may fail to terminate. Moreover, their general model is quite expensive. The paper discusses a static analysis based on abstract interpretation as a starting point towards an efficient implementation.

Prolog has also been used to navigate code [18], to define AOP as logic meta programming [8], to define aspect specific languages [5], and to define more robust pointcut definitions [17]. However these work focus on context in the abstract syntax tree (e.g. nested definitions) rather than context in the execution.

EAOP [9] provides a pointcut language based on regular expressions to define sequences of execution events. Sequences can be of arbitrary length, but they do not share references (i.e., express dependencies). When variable sharing between pointcuts is introduced [10] chains of bound length can be

```

1 aspect ColoredConnectedFigures {
2   before(ConnectionFigure cf,Connector c):
3     execution(void ConnectionFigure.connectStart(Connector)) && target(cf) && args(c) {
4       CONNECTSTART(cf,c);
5     }
6   before(ConnectionFigure cf):
7     execution(void ConnectionFigure.disconnectStart()) && target(cf) {
8       ! CONNECTSTART(cf,-);
9     }
10  before(ConnectionFigure cf,Connector c):
11    execution(void ConnectionFigure.connectEnd(Connector)) && target(cf) && args(c) {
12      CONNECTEND(cf,c);
13    }
14  before(ConnectionFigure cf):
15    execution(void ConnectionFigure.disconnectEnd()) && target(cf) {
16      ! CONNECTEND(cf,-);
17    }
18  after(Figure f) returning(Connector c):
19    execution(Connector.new(Figure)) && args(f) {
20      CONNECT(f,c);
21    }
22  // rules for connected figures
23  CONNECT(f1,c1) && CONNECTSTART(cf,c1) && CONNECTEND(cf,c2) && CONNECT(f2,c2) => CONNECTED(f1,f2)
24
25  CONNECTED(f1,f2) && CONNECTED(f2,f3) => CONNECTED(f1,f3)
26
27  CONNECTED(f1,f2) => CONNECTED(f2,f1)
28
29  // repaint propagation
30  around(Figure f,Object value):
31    execution(public void Figure.setAttribute(String, Object)) && target(f)
32    && args("FillColor",value) {
33      for f2 in CONNECTED(f,f2)
34        do f2.setAttribute("FillColor",value);
35    }
36 }

```

Figure 4: Color Updating of Connected Figures in JHotDraw as a Relational Aspect

defined. On the other hand, EAOP relies on finite state automata which cannot be simulated with Datalog.

Trace matching [1] extends AspectJ with sequences of join-points. Pointcuts in the sequence definition can share variables as in a Datalog rule. However, sequences are restricted to regular expressions, and trace matching does not allow recursive definitions (e.g., the transitive closure of connected figures in JavaHotDraw). Finally, when events can occur in different order, the different sequences must be enumerated.

Another approach to context passing relies on implicit context [27]. A component is defined by its boundary and filters are responsible for intercepting and modifying incoming and outgoing messages. A specialized language (e.g. `FindLastCallTo`, `FindLastCallToFrom`) is used to query a call history. Complex queries can be defined in a programmatic way. This work can be seen as a specialization of composition filters [4], which provide pattern matching to select filters but no history mechanism (although it can be programmed). These approaches are more expressive but less declarative than ours: they partially rely on a general purpose programming language.

Other work focuses on *structural* relationships (i.e., fields of objects) only. The most obvious example is object-oriented data-base systems [3]. These applications consider much more than relations: integrity constraints, concurrency, se-

curity...DemeterJ [24, 26] also defines relationships in a graph of objects. It allows defining reusable traversal strategies that implements the visitor design pattern. The intermediate structure (i.e., links) between two points of interest in the graph do not have to be specified. Finally, different static analyses have been developed in order to guarantee a regular structure to the graph of objects. Such analyses can be used at the programming-level [22, 23, 14], or at the design-level [16]. All these proposals deal only with the object graph but do not enables to define relations that do not exist in the store.

4. CONCLUSION

In this article we have proposed relational aspects to generalize context passing with Datalog as a pointcut language. The benefits of our approach are:

- it is non invasive. The programmer does not modify the base program, but she specifies relations. Relational aspects do not modify the base program semantics: they maintain relations and do not change termination properties (all Datalog programs terminate [6]). (Of course, when a relation is queried, an advice can modify the base program semantics).
- it is declarative. The programmer does not have to worry about the representation of relations. Relations have no orientation and can be queried different ways.

The programmer does not have to worry about the order of the method calls: rules do not impose an order for tuples creation. (Of course the programmer has to worry about the order of the method calls that delete tuples.) Rules are automatically applied to update relations as needed. The programmer does not have to worry about cycles (Datalog semantics is based on a fixpoint).

- it is expressive. Relational aspects reify and make persistent relations. These relations crosscut the trace rather than the stack. A chain of relations can be of arbitrary length. Relations are not restricted to pairs. We have successfully applied our approach to different applications (e.g., a constraint solver, a graphical editor).

This work offers many opportunities for future work. First, we have implemented a simple Datalog interpreter in order to test the examples of this paper. We should now consider using industrial Datalog engines such as Jess[19] in order to conduct more experiments. For instance, in JHotDraw observers [15] could be replaced by relational aspects.

5. REFERENCES

- [1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA '05*. ACM Press, 2005.
- [2] AspectJ Site. Aspect-oriented programming in Java with AspectJ, 2001. <http://www.parc.com/research/projects/aspectj>.
- [3] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The OO database system manifesto. In *DOOD '89*, pages 223–240, 1989.
- [4] L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. *Com. of the ACM*, 44(10):51–57, 2001.
- [5] J. Brichau, K. Mens, and K. D. Volder. Building composable aspect-specific languages. In *GPCE'02*, volume 2487 of *LNCS*, pages 110–127. Springer-Verlag, 2002.
- [6] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog. *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.
- [7] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectC to improve the modularity of path-specific customization in OS code. In *FSE'01*. ACM Press, 2001.
- [8] K. De Volder and T. D'Hondt. Aspect-oriented logic meta programming. In *Reflection'99*, volume 1616 of *LNCS*, pages 250–272. Springer-Verlag, 1999.
- [9] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *GPCE'02*, volume 2487 of *LNCS*, pages 173–188. Springer-Verlag, 2002.
- [10] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04*. ACM Press, 2004.
- [11] R. Douence and N. Jussien. Non-intrusive constraint solver enhancements. In *First AOSD workshop on ACP4IS*. University of Twente, 2002.
- [12] R. Douence and L. Teboul. A pointcut language for control-flow. In *GPCE'04*, volume 3286 of *LNCS*. Springer-Verlag, 2004.
- [13] P. Fradet and D. L. Metayer. Shape types. In *POPL '97*, pages 27–39. ACM Press, 1997.
- [14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP'93*, volume 707 of *LNCS*, pages 406–431. Springer-Verlag, 1993.
- [15] Y. Guéhéneuc. A systematic study of UML class diagram constituents for their abstract and precise recovery. In *APSEC '04*. IEEE, 2004.
- [16] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03*, pages 60–69, New York, NY, USA, 2003. ACM Press.
- [17] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *AOSD '03*, pages 178–187, New York, NY, USA, 2003. ACM Press.
- [18] Jess Site. Jess, the rule engine for the java platform. <http://herzberg.ca.sandia.gov/jess/>.
- [19] N. Jussien. The versatility of using explanations within constraint programming. Habilitation thesis of Université de Nantes, 2003.
- [20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *ECOOP'01*, volume 2072 of *LNCS*. Springer-Verlag, 2001.
- [21] N. Klarlund and M. Schwartzbach. Graph types. In *POPL '93*, pages 196–205. ACM Press, 1993.
- [22] A. Moller and M. Schwartzbach. The pointer assertion logic engine. In *PLDI '01*, pages 221–231. ACM Press, 2001.
- [23] D. Orleans and K. Lieberherr. DJ: Dynamic adaptive programming in Java. In *Reflection '01*, volume 2192 of *LNCS*, pages 73–80. Springer Verlag, 2001.
- [24] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP'05*, volume 3586 of *LNCS*, pages 214–240. Springer-Verlag, 2005.
- [25] M. Shonle, K. Lieberherr, and A. Shah. Xaspects: An extensible system for domain specific aspect languages. In *OOPSLA '03*, pages 28–37. ACM Press, 2003.
- [26] R. Walker and G. Murphy. Implicit context: easing software evolution and reuse. In *FSE '00*, pages 69–78. ACM Press, 2000.