Aspectual Source Code Analysis with GASR

Johan Fabry
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile
Santiago, Chile
http://pleiad.cl

Coen De Roover
and Viviane Jonckers
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
http://soft.vub.ac.be

***Abstract*—To be able to modularize crosscutting concerns, aspects introduce new programming language features, often in a new language, with a specific syntax. These new features lead to new needs for source code analysis tools, resulting in the requirement for a general-purpose aspectual source code analysis tool. Ignoring this requirement has led to a nontrivial duplication of effort in the aspect-oriented software development community. This is because all code analysis efforts that we are aware of have either built ad-hoc analysis tools or were performed manually. In this paper we present GASR: a source code analysis tool in the tradition of logic program querying that reasons over ASPECTJ source code. By hooking into the IDE plugins for ASPECTJ, GASR provides a library of predicates that can be used to query aspectual source code. We demonstrate the use of GASR by automating the recognition of a number of previously identified aspectual source code assumptions. We then detect assumption instances on two well-known case studies that were manually investigated in the earlier work. In addition to finding the already known aspect assumptions, GASR encounters assumption instances that were overlooked before.**

***Keywords*—Aspect Oriented Programming, Logic Program Querying, Aspectual Assumptions**

I. INTRODUCTION

Aspects are a means to modularize *cross-cutting concerns*: concerns whose implementation is spread throughout different modules of the system under construction. Aspects are a new kind of module that encapsulate, in addition to their behavior, when this behavior needs to be invoked, *i.e.,* also define a kind of implicit invocation of their behavior.

To perform Aspect-Oriented Programming, new programming languages have been proposed that are usually extensions of existing OOP languages. These extensions then consist of language features that allow for the specification of these new modules, and most importantly their implicit invocation conditions, known as *pointcuts*. As a result of this, existing source code analysis tools for these OOP languages are incapable to correctly treat these aspects in their reasoning. Firstly existing analyses may be incorrect and, secondly, analyses that specifically consider the aspectual properties of the source code are absent. Considering the first case: as aspects modify the control flow of the program, source code analysis should take these changes into account when reasoning over properties of the code, where appropriate. As for the second case, the extensions made by aspect languages are usually nontrivial. This creates an entirely new class of analyses that take into account the aspectual nature of the code and how these features are used and interact (*e.g.,* [1], [2], [3], [4], [5], [6], [7], [8], [9]). To the best of our knowledge, all of these kinds of analyses have been made on an ad-hoc basis, customized specifically to the analysis task being performed. As a result there has been a considerable duplication in development effort of these analyses. Moreover these are not customizable to the actual software under analysis, *e.g.,* to automatically remove one kind of known false positives from the results.

978-1-4673-5739-5/13/$31.00

We state that there is a need for a general-purpose source code reasoner for aspects. It should ease the definition of multiple kinds of source-code analysis over aspect-oriented source code and also be tailorable to the task at hand by the user. To address this need we built GASR and we present it in this paper.

This paper contains the following contributions:

- It argues for the need for a general-purpose source code analysis tool that is aware of aspects.
- It presents the logic program querying tool GASR, the first such analysis tool, and discusses its implementation along with its library of logical predicates.
- It shows how GASR can be used to automatically verify a subset of previously published inter-aspect assumptions [10], implementing part of the future work of that publication.

The structure of this text is as follows: next we provide the problem statement of the paper, arguing for the need for GASR. In Section III we give an overview of related work, showing that existing analyses have been ad-hoc. We then present GASR in Section IV, discussing its implementation and a selection of its library of logical predicates. This is followed, in Section V, by an illustration of the usefulness of GASR by realizing detection of inter-aspect assumptions, as previously identified in [10]. The paper then briefly discusses threats to validity in Section VI before providing conclusions and avenues for possible future work.

II. PROBLEM STATEMENT

To enable the modular specification of crosscutting concerns, aspects encapsulate both their behavior as well as the invocation conditions for this behavior. This gives rise to new language features and terminology, which in turn requires new features for a source code reasoner.

A. Terminology and Language Features

Broadly put, an aspect contains two parts: its behavior, specified in a number of *advice*, and the invocation specifications for this advice, denoted in *pointcuts*. Advice are linked to pointcuts, and whenever a pointcut matches the linked advice is invoked. Conceptually, to match pointcuts each execution step of the software is reified as a *join point* and pointcuts are predicates over join points. The work of performing join point reification, passing them to all pointcuts, and running the associated advice if a pointcut matches is performed by the *aspect weaver*. Implementation strategies for aspect weavers vary from source-code preprocessing to aspect-aware virtual machines. A last item of terminology is the *join point shadow* for a join point: the piece of source code whose execution produced that join point.

We now show different language features that the prototypical aspect-oriented language places at the programmers' disposal, as an indication towards the possible complexity of aspectual source code. The example language is ASPECTJ [11], arguably the best-known and most-used aspect language. ASPECTJ is an extension of Java that introduces aspect features using a specific syntax. We now briefly touch on the different aspectual language features of ASPECTJ, starting with aspects: Aspect declarations are similar to class declarations and are declared using the aspect keyword. Aspects can contain methods and fields, but only one zero-argument constructor. The latter is because aspects cannot be manually instantiated, the weaver performs this when needed (typically aspects are singletons). Aspects may be abstract, extend classes and abstract aspects, and implement interfaces.

Pointcuts in ASPECTJ are a new sort of member declaration that use the pointcut keyword, and have the standard visibility and inheritance semantics. Pointcuts have a body, unless they are declared as abstract. Abstract pointcuts can only be contained in an abstract aspect. The body of a pointcut is a logical combination of pointcut expressions or a primitive pointcut expression. Primitive pointcut expressions firstly specify the kind of pointcut: an execution of a method, a call of a method, getting or setting a field, and so on. Secondly, they provide a pattern that may match on that kind of execution step of the software, *e.g.,* a signature of a method. In this pattern wildcards may be used to generalize over names as well as types.

An example of a quite drastic pointcut that uses a pattern is below: a pointcut named allFoo that matches on the method calls of all methods of the class Foo, irrespective of the return type and number of parameters.

```
1public pointcut allFoo() : call(* Foo.*(..));
```

Advice are similar to methods in that they declare a body of code and have parameters. They differ firstly in that they do not have a name, but instead declare that they need to be invoked before, after, or instead of (around) the join point. They link to a pointcut by providing the pointcut name, or a pointcut body (known as using an anonymous pointcut).

Lastly, aspects may also modify the type hierarchy and add *inter-type declarations*. In the former the aspect declares that given classes or aspects extend or implement other classes or interfaces. In the latter the aspect adds fields or methods to other classes, similar to what is allowed in Open Classes [12].

B. Classic Example: Aspect Reentrancy

As a first, brief, example of a concrete need for aspect-specific source code reasoning we now present a classic example of an ASPECTJ antipattern regarding reentrancy. We include it here as it is important, yet simple enough to be briefly explained. Consider the following as a token of its importance: the ASPECTJ documentation 'pitfalls' section[1] contains just this one example, and no other.

```
1aspect Boom {
2   before(): call(* *(..)) {
3     System.out.println("before"); }}
```

The aspect above declares one advice, with a pointcut body that matches *on all method calls in the program*. The behavior of the advice is to make a method call to the System.out.println method. The pointcut matches on all method calls, hence also this call, hence before the method is called the advice body is again executed, leading to an infinite loop.

The antipattern in the above example can be easily detected by an aspect-aware source code reasoner: there is a possibility for infinite application of an advice when the join point shadows of the associated pointcut are contained in this advice.

C. Problem: A New Reasoning Need

If we consider aspects as simply a means to achieve behavior subject to implicit invocation with implicit announcement [13], it may seem that the need for source code reasoning over aspects is simple. Since in this view aspects essentially are for altering the control flow of the running application, existing source code reasoners just need to be extended to take this control flow into account.

Aspects however go beyond the above as they introduce multiple aspectual language features that interact with the non-aspectual language features as well as among themselves. An example of the former is that aspects may change the class hierarchy of the program. As an example of the latter consider a pointcut named abstractpc, defined as an abstract pointcut in a root aspect Root and also used by an advice of Root. abstractpc may be concretized in a child-aspect Child of Root. It may also be concretized again in an aspect Grandchild that is a child of Child, *i.e.,* a grandchild of Root. The definition of the actual pointcut that is used for the advice in Root is the lowest in the hierarchy [10], *i.e.,* the re-concretization in Grandchild.

As a result, in addition to the classic example of Sect. II-B, many possible issues in aspectual code have been separately identified. We provide three examples. First are aspects assuming specific properties of other aspects to be present [10], which we will discuss in more detail in Section V. Second is the problem of pointcuts that are slightly or subtly incorrect [1], as a result these fail to match the intended join points, or match unintended join points. Third is the fragility of pointcuts when the software evolves [3], [4], in this case pointcuts end up being broken due to changes in the program that were made due to its evolution. We find it remarkable that for these three examples no single source code reasoner can yet be used to detect all of these issues such that they can be revealed using, *e.g.,* a bad smells detection tool.

[1] http://www.eclipse.org/aspectj/doc/next/progguide/pitfalls.html

Also, multiple aspectual design patterns have been presented [14], [15], yet no mining of these patterns with a source code reasoner have been documented. A well-known example is the Wormhole [15]: an aspect intervenes in one part of the control flow to store the value of a specific parameter or variable, and in a second part retrieves this value and injects it back in the control flow. It is as if the value passed through a wormhole that lies between both parts. Given that a pattern is a template, there may be various variations on this template, and various ways in which these are instantiated. Hence an analysis tool to discover such patterns or to verify their correct use, *e.g.,* if the stored value is modified before it is injected, would need to be tailorable to the case at hand.

From the above, we conclude that there is a need for multiple kinds of source-code analysis over aspect-oriented software that are general enough such that they can be used for multiple kinds of analysis and moreover are adaptable such that they can be tailored to the task at hand. In other words, we need a general-purpose source code reasoner for aspects. With this reasoner we would then be able to, *e.g.,* automatically identify aspectual assumptions in code, write a bad smells tool that can reveal errors as in Sect. II-B, or detect incorrect use of the Wormhole pattern.

III. Related Work

To the best of our knowledge, there is no general-purpose aspect source code analysis tool. Directly related work consists of specific ad-hoc analyses made, and indirectly related is work on code comprehension of aspectual source code.

Getting pointcuts correct can be a hard task [1], and as a result of this, pointcuts have been the focus of various tracks of research that include code reasoning. Notable early work is on PointcutDoctor [1], a tool that provides special-purpose reasoning over pointcuts to establish near matches of pointcuts as well as the reasons why a given shadow matches, or does not match a specific pointcut. Related to this is the fragility of pointcuts, as mentioned above. The most recent work is on pointcut rejuvenation [2]. New code that is added as the software evolves may also need to be captured by the existing pointcuts, *i.e.,* they need to be changed. A custom analysis is developed that suggests changes to pointcuts when needed. Earlier work in this area [3], [4] also used custom analyses.

Yet reasoning about aspectual source code is not limited to pointcuts only. For example, ITDVisualizer [5] is a tool that supplies an analysis of the impact of intertype declarations. It shows how they impact method lookup, and identifies how code entities are shadowed by intertype declarations. XFindBugs [6] is a tool that uses static analysis to find potential bugs in aspectual source code. It defines a catalog of multiple bug patterns for aspect-oriented features, and implements a set of bug detectors on top of the FindBugs analysis framework[2]. Last but not least, the work on the Ajana analysis framework [7] for source-code-level interprocedural dataflow analysis yields a control- and data-flow program representation for aspectual source code. Considering this representation it proposes an object effect analysis and a dependency analysis. Again all of the above tools use a custom reasoner to provide the analysis.

[2]http://findbugs.sourceforge.net/

Complementary to the above, code comprehension tools for aspectual source code also include some form of ad-hoc reasoning to be able to display their specific comprehension aids. We highlight two such tools: the AJDT and AspectMaps.

The AspectJ Development Toolkit (AJDT) [8] is arguably the most mature, feature-rich and best known tool suite for AOP. It consists of a set of plug-ins to the Eclipse IDE that add code comprehension features, amongst others. It provides a "Cross References" view that, when editing an aspect or class, shows a summary of the join point shadows or advice that apply, respectively. In the code editor, at each join point shadow, gutter markers are present that reveal information about the advice. AJDT also provides for a visualization of the source code, but this feature has been superseded by other aspectual visualizations, the most recent of which is AspectMaps [9], [16]. AspectMaps is a visualization tool that shows where in the code aspects apply. Of all aspect visualizations, AspectMaps shows the most information about the source code [9]. Moreover, by using a selective structural zoom, it ensures a scalable visualization from package level all the way down to method level. At this finest granularity it shows exactly where advice apply, the order of advice execution at one shadow and whether the advice has any runtime invocation conditions.

A common thread in all the above work is that the required source code analysis is provided ad-hoc, entailing a significant duplication of effort. If a general-purpose aspect-oriented source code reasoner would have existed, this duplication of effort might have been avoided.

IV. Querying AspectJ Programs using Gasr

We introduce Gasr (General-purpose Aspectual Source code Reasoner) as a tool for answering user-specified questions about the structure as well as the behavior of an aspect-oriented program. Examples range from "*which pointcut definitons are overridden in a subtype?*" over "*which pointcuts have a join point shadow in an advice?*" to "*can these advices be executed consecutively?*". Such questions have to be specified as a logic query of which the conditions quantify over the program's source code. The expressiveness of the logic paradigm has been shown to facilitate specifying the characteristics of sought after code. Once specified in a logic program query, retrieving source code elements that exhibit these characteristics is left to the querying tool. This relieves users of having to implement an imperative search themselves. As such, Gasr is a tool in the tradition of logic program querying. Other examples include CodeQuest [17], PQL [18] and Soul [19].

Gasr owes its query language to the core.logic[3] port to Clojure of miniKanren [20], and its IDE integration to the Ekeko[4] Eclipse plugin. The latter enables launching and scheduling program queries, as well as inspecting the solutions to a query and associating warning markers with them — actually building upon our earlier Eclipse plugin suite for program querying [19], [21].

[3]https://github.com/clojure/core.logic

[4]https://github.com/cderoove/damp.ekeko/

A. Launching Program Queries

Queries can be launched from a read-eval-print loop using the `ekeko*` special form. It takes a vector of logic variables, each denoted by a starting question mark, as its first argument and this is then followed by a sequence of logic goals:

```
1 (ekeko* [?x ?y]
2   (contains [1 2] ?x)
3   (contains [3 4] ?y))
```

The binary predicate `contains/2`, used by both goals, holds if its first argument is a collection that contains the second argument. Solutions to a query consist of the different bindings for its variables such that all logic goals succeed. Internally, the logic engine performs an exploration of all possible results, using backtracking to yield the different bindings for logic variables. The four solutions to the above query consist of bindings `[?x ?y]` such that `?x` is an element of vector `[1 2]` and `?y` is an element of vector `[3 4]`: `([1 3] [1 4] [2 3] [2 4])`.

Logic variables have to be introduced explicitly into a lexical scope. Above, the `ekeko*` special form introduced two variables into the scope of its logic conditions. Additional variables can be introduced through the `fresh` special form:

```
1 (ekeko* [?x]
2   (differs ?x 4)
3   (fresh [?y]
4     (equals ?y ?x)
5     (contains [3 4] ?y)))
```

The above query has but one solution: `([3])`. Indeed, `3` is the only binding for `?x` such that all goals suceed. The `differs/2` goal on line 2 imposes a disequality constraint such that any binding for `?x` has to differ from `4`. The `equals/2` goal on line 4 requires `?x` and the newly introduced `?y` to unify.

Finally, new predicates can be defined as regular Clojure functions that return a logic goal. As such, the aforementioned special forms give rise to an *embedding* of logic programming in a functional language.

```
1 (defn contains+ [?c ?e]
2   (conde [(contains ?c ?e)]
3          [(fresh [?x]
4            (contains ?c ?x)
5            (contains+ ?x ?e)])))
```

Here, the special form `conde` returns a goal that is the disjunction of one or more goals. The newly defined predicate `contains+` therefore succeeds for `?e` that reside at an arbitrary depth within a collection `?c`.

Note that an idiomatic Prolog definition of the above would consist of two rules that define the same predicate: one for the base case and one for the recursive case, thus creating an implicit choice point. By relying on function definition, the above implementation has to make such choice points explicit.

B. The Predicate Library of GASR

To enable querying ASPECTJ programs, we have developed a library of predicates that can be used in EKEKO queries. For instance, solutions to the following query correspond to instances of the aspect reentrancy example described in Section II-B:

```
1 (ekeko* [?aspect ?advice]
2   (fresh [?shadow]
3     (aspect-advice ?aspect ?advice)
4     (advice-shadow ?advice ?shadow)
5     (shadow-enclosing ?shadow ?advice)))
```

Upon backtracking, the goal on line 3 successively binds `?advice` with each advice of an aspect `?aspect` —which is also bound successively to each aspect known to the ASPECTJ weaver. The goal on line 4 binds `?shadow` to one of the join point shadows of this advice, while the goal on line 5 requires `?advice` to unify with the immediately enclosing source code entity of `?shadow`. Hence, `?advice` will be bound to an advice that advices itself, *i.e.,* a possible infinite loop. Note that, by convention, the names of predicates that reify an n-ary relation consist of n components separated by a `-`, each describing an element of the relation. Also, vertical bars `|` separate words within the description of a single component.

The predicates used in the above query concern the structure of the woven ASPECTJ program. In contrast, the predicates below concern possible behavior of the program at run-time. Its solutions correspond to possible instances of the wormhole pattern described in Section II-C:

```
1 (ekeko* [?aspect ?advice|entry ?advice|exit ?field]
2   (aspect-advice ?aspect ?advice|entry)
3   (type-field ?aspect ?field)
4   (advice|writes-field ?advice|entry ?field)
5   (differs ?advice|exit ?advice|entry)
6   (aspect-advice ?aspect ?advice|exit)
7   (advice|reads-field ?advice|exit ?field)
8   (advice-reachable|advice ?advice|entry ?advice|exit))
```

The first goal binds `?advice|entry` to an advice that will serve as the entry point of the wormhole `?aspect`. Lines 3–4 therefore ensure that this advice writes to a `?field` defined in the same aspect. Lines 5–6 require this aspect to feature a different `?advice|exit` that will serve as the exit point of the wormhole. As such, the exit advice has to read from the field written to by the entry advice (line 7). Note how multiple occurrences of a logic variable link these goals together. The final goal conservatively ensures that there might be an execution of the woven program in which `?advice|exit` is executed after `advice|entry`.

We have developed a comprehensive library of logic predicates, which we do not discuss in full here. Instead, Table I and Table II list representative predicates that reify structural resp. behavioral relations between ASPECTJ source code entities. We refer to the online documentation[5] for an overview of the complete predicate library. The remainder of this section discusses the highlights of its implementation.

1) Predicates Reifying Structural Relations: The predicates listed in Table I reify the structural relations between the source code entities of an ASPECTJ program (*e.g.,* types and their members, aspects and their pointcut definitions, advices and their shadows). To this end, their implementation consults the domain model maintained by the ASPECTJ weaver.

EKEKO supports calling out to Java from within a logic goal. This obviates the need to convert the weaver's domain objects to logic facts. Instead, they are kept as instances

[5] https://github.com/cderoove/damp.ekeko.aspectj

Predicate	Reified Relation
`(type ?type)`	Of all types known to the weaver (*i.e.,* aspects, classes, interfaces, enums, *etc*).
`(type-declaredsuper ?type ?super)`	Between a type and its direct declared superclass or superaspect.
`(type-declaredinterface ?type ?interface)`	Between a type and one of the interfaces it declares to be implementing or extending directly.
`(type-super+ ?type ?super)`	Between a type and one of its direct or indirect super types (classes, aspects as well as interfaces), including those that stem from an intertype declaration.
`(type-method ?type ?method)`	Between a type and one of its declared methods.
`(type-method+ ?type ?method)`	Between a type and one of its declared or inherited methods. Does not include methods stemming from intertype declarations.
`(aspect ?aspect)`	Of all aspects known to the weaver. Subrelation of `type/2`.
`(aspect-pointcutdefinition ?aspect ?pointcutdefinition)`	Between an aspect and one of its declared pointcut definitions.
`(aspect-advice ?aspect ?advice)`	Between an aspect and one of its declared advice.
`(aspect-intertype ?aspect ?intertype)`	Between an aspect and one of its intertype member declarations.
`(aspect-declare ?aspect ?declare)`	Between an aspect and one of its `declare` declarations (*e.g.,* parents, precedence).
`(pointcutdefinition-pointcut ?pointcutdefinition ?pointcut)`	Between a non-abstract pointcut definition and its pointcut.
`(pointcutdefinition-name ?pointcutdefinition ?name)`	Between a pointcut definition and its name.
`(pointcutdefinition|abstract ?pointcutdefinition)`	Of abstract pointcut definitions. Sub-relation of `pointcutdefinition/1`.
`(advice|before ?advice)`	Of `before` advices. Sub-relation of `advice/1`.
`(advice-pointcut ?advice ?pointcut)`	Between an advice and its pointcut. The latter either an anonymous pointcut, or a pointcut definition.
`(advice-pointcutdefinition ?advice ?pointcutdefinition)`	Between an advice and the concrete pointcutdefinition its name resolves to (*i.e.,* overrides of possibly abstract pointcutdefinitions are taken into account).
`(advice-shadow ?advice ?shadow)`	Between an advice and one of its join point shadows.
`(shadow-enclosing ?shadow ?enclosing)`	Between a shadow and its immediately enclosing entity or the entity itself for entity shadows. This entity can be a class, aspect, enum, method, intertype method, advice, *etc* ...
`(shadow-ancesor|type ?shadow ?type)`	Between a shadow and its first enclosing type entity (*e.g.,* aspect, class, enum).
`(intertype-member-target ?intertype ?member ?type)`	Between an intertype declaration, the member (*i.e.,* field, method or constructor) it declares, and the type to which this member is added.
`(declare|parents ?declare)`	Of `declare parents` declarations. Subrelation of `declare/1`.
`(declare|parents-target-parent ?declare ?target ?parent)`	Between a `declare parents` declaration, one of the target types matching its pattern and the corresponding super type.
`(declare|precedence ?declare)`	Of `declare precedence` declarations. Subrelation of `declare/1`.
`(aspect|dominates-aspect ?daspect ?saspect)`	Of actual domination relations between aspects that result from `declare precedence` declarations.

TABLE I. REPRESENTATIVE PREDICATES CONCERNING STRUCTURE.

Predicate	Reified Relation
`(advice|reads-field ?advice ?field)`	Between an advice and one of the fields it reads from.
`(advice|writes-field ?advice ?field)`	Between an advice and one of the fields it writes to.
`(advice-reachable|advice ?advice1 ?advice2)`	Between an advice and another advice such that the latter may be executed after the former. Concretely, this is the relation of two successive advices on a path through the inter-procedural control flow graph of the woven program.
`(field-soot|field ?field ?soot)`	Between a field and the SOOT field that represents its implementation.
`(advice-soot|method ?advice ?soot)`	Between an advice and the SOOT method that represents its implementation.
`(intertype|method-soot|method ?itmethod ?soot)`	Between a method declared by an intertype declaration and the SOOT method that represents its implementation.
`(soot|method-soot|unit ?method ?unit)`	Between a SOOT method and one of the units in its body. These correspond to instructions in SOOT's JIMPLE intermediate representation [22] of the woven program.
`(soot|unit|reads-soot|valuebox ?unit ?value)`	Between a SOOT unit and one of the values (*e.g.,* parameters, field references, expressions...) it reads from.
`(soot|unit|writes-soot|valuebox ?unit ?value)`	Between a SOOT unit and one of the values it writes to.
`(icfg|main-start ?icfg ?icfg|start)`	Between the inter-procedural control flow graph of the woven program and its starting node.
`(icfgnode-unit ?node ?unit)`	Between a node of the inter-procedural control flow graph and a SOOT unit.
`(icfgnode-method ?node ?method)`	Between a node of the inter-procedural control flow graph and the SOOT method in which it resides.
`(icfgnode-stack?node ?stack)`	Between a node of the inter-procedural control flow graph and the (finite) configuration of the call stack at the time it was encountered during a traversal.
`(path ?icfg ?start ?end [`$v_1 \ldots v_n$`] `$q_1 \ldots q_n$`)`	Of inter-procedural control flow graphs `?icfg` in which there exists a path from `?start` till `?end` that is of the form described by the regular path expression $q_1 \ldots q_n$. Here q is one of the regular path primitives provided by the QWAL library [23]: `q=>` skips a single node, `q=>*` skips zero or more nodes, and `q=>+` skips one or more nodes on the path. Primitive `qcurrent` evaluates logic goals against the current node on the path, possibly involving one of the $v_1 \ldots v_n$ logic variables.

TABLE II. REPRESENTATIVE PREDICATES CONCERNING BEHAVIOR.

of various org.aspectj.weaver classes. The binary predicate `aspect-pointcutdefinition/2`, *e.g.,* is as follows:

```
1 (defn aspect-pointcutdefinition [?aspect ?pcdef]
2   (fresh [?pcdefs]
3     (aspect ?aspect)
4     (equals ?pcdefs (.getDeclaredPointcuts ?aspect))
5     (contains ?pcdefs ?pcdef)))
```

The predicate reifies the relation between an aspect and one of its own, non-inherited pointcut definitions. The goal on line 3 ensures that `?aspect` is bound to the weaver's representation of an aspect (*i.e.,* an instance of ResolvedType). This enables the goal on line 4 to unify `?pcdefs` with the result returned by method getDeclaredPointcuts() on the binding of `?aspect`. Upon backtracking, the goal on line 5 will therefore successively unify `?pcdef` with each of the elements of the returned collection of ResolvedPointcutDefinition instances.

2) Predicates Reifying Behavioral Relations: The predicates listed in Table II reify control flow and data flow relations between the source code entities of the *woven* ASPECTJ program. While the former concerns the order in which instructions may be executed at run-time, the latter concerns the values these instructions may operate upon.

The predicates at the top of Table II reify behavioral relations between elements that stem from the weaver's domain model. These can be combined with the structural predicates of Table I. For example, solutions to the following consist of an advice and a type of which the advice modifies a field:

```
1 (ekeko* [?advice ?type]
2   (fresh [?field]
3     (advice|writes-field ?advice ?field)
4     (type-field ?type ?field)))
```

These predicates are implemented themselves in terms of predicates that quantify over static analysis results provided by the SOOT [22] analysis framework (third row in Table II) and predicates that link both sources of information together (second row in Table II). For instance, the binary predicate `advice|writes-field/2` is implemented as follows:

```
1 (defn advice|writes-field [?advice ?field]
2   (fresh [?soot|method ?soot|field ?soot|unit
3           ?vbox ?value]
4     (advice-soot|method ?advice ?soot|method)
5     (field-soot|field ?field ?soot|field)
6     (soot|method-soot|unit ?soot|method ?soot|unit)
7     (soot|unit|writes-soot|valuebox ?soot|unit ?vbox)
8     (soot|valuebox-soot|value ?vbox ?value)
9     (succeeds (instance? soot.jimple.FieldRef ?value))
10    (equals ?soot|field (.getField ?value))))
```

The goal on line 4 retrieves SOOT's representation of the method that represents the weaver's advice `?advice` in the woven program. The goal on line 5 does the same for the weaver's field `?field`. The remaining goals use predicates that reify relations between SOOT elements only. Upon backtracking, the goal on line 6 will successively unify `?soot|unit` with one of the units in the body of `?soot|method`. These correspond to instructions in SOOT's JIMPLE intermediate representation [22] of the woven program. Lines 7–10 ensure that this unit writes to the SOOT field `?soot|field` that represents the weaver's field `?field` in the woven program. Note that the final goal calls out to SOOT to resolve a field reference to the referenced field —possible because we forego a conversion to logic facts.

Predicates such as `advice-reachable|advice/2`, which reifies the relation between an advice and another advice such that the latter may be executed after the former, require more detailed information about the woven program. They are hence implemented in terms of predicates that quantify over the paths through an inter-procedural control flow graph of the woven program (fourth row in Table II). We compute this graph by linking the intra-procedural control flow graphs of callers and callees using the results of SOOT's points-to analysis, *i.e.,* the demand-driven, context-sensitive version by Sridharan et al. [24]. We refer to the online documentation of EKEKO for behavioral predicates that reify may-alias and must-alias dataflow relations between SOOT values based on this analysis.

To summarize: the possible methods an invocation may resolve to are determined using a compile-time approximation of the dynamic type of its receiver, *i.e.,* the types of the objects in its points-to set, rather than its static type — which is more precise. Note that multiple call sites result in control flow splits at the exit points of callees for link-based whole-program graphs. Our graph traversal predicates therefore take care not to follow unrealizable paths, without endangering termination (*i.e.,* a *finite* call stack ensures that successors of a method's exit node agree with an earlier method invocation).

Of the graph traversal predicates at the bottom of Table II, `path/n` is of special interest as it embodies the implementation of parametric regular path expressions [25], [26] in EKEKO (which we have applied in earlier work to query the history of versioned software [23]). Regular path expressions are an intuitive formalism for quantifying over the paths through a graph. They are akin to regular expressions, except that they consist of logic goals to which regular expression operators have been applied. Rather than matching a sequence of characters in a string, they match paths through a graph along which their logic goals succeed. This is illustrated by the implementation of predicate `advice-reachable|advice/2` below:

```
1 (defn advice-reachable|advice [?advice1 ?advice2]
2   (fresh [?s|method1 ?s|method2
3           ?icfg ?icfg|start ?icfg|end]
4     (advice-soot|method ?advice1 ?s|method1)
5     (differs ?advice1 ?advice2)
6     (advice-soot|method ?advice2 ?s|method2)
7     (icfg|main-start ?icfg ?icfg|start)
8     (path ?icfg ?icfg|start ?icfg|end []
9       (q=>*)
10      (qcurrent [?n]
11        (icfgnode-method ?n ?s|method1))
12      (q=>+)
13      (qcurrent [?n]
14        (icfgnode-method n ?s|method2)))))
```

The goals on lines 4–6 of quantify over two distinct advices and their corresponding SOOT methods in the woven program. Line 7 unifies `?icfg` with an inter-procedural control flow graph that starts at the `main()` method of the woven program. The goal on line 8 succeeds if there is a path through this graph from node `?icfg|start` to `?icfg|end` that is of the form described by the regular path expression in its body: zero or more non-distinct nodes (*i.e.,* nodes against which no logic goals have to succeed) (line 9), followed by one node that resides in the SOOT method corresponding to `?advice1` (lines 10-11), followed in turn by one or more non-distinct nodes (line 12), concluded by a node that resides in the SOOT method corresponding to `?advice2` (line 13).

Note that a similar regular path expression can be used to warn about possibly incorrect implementations of the wormhole pattern described in Section II-C. These are characterized by an execution path on which the wormholed field is written to inbetween the entry and exit advice.

V. DETECTING ASPECT ASSUMPTIONS WITH GASR

As an illustration of the usefulness of GASR we now show how it can be used to implement detection of developers' assumptions about aspect usage, effectively extending the "Aspect Assumptions" work of Zschaler and Rashid [10]. For brevity, in the rest of this section we will refer to this work as AA. For AA, Zschaler and Rashid have studied three nontrivial aspectual systems to discover the assumptions made by the different modules about the functionality, presence and implementation of other modules. The authors assert that assumptions that aspects make about the system are "particularly critical" [10] because of the cross-cutting nature of aspects as well as their implicit invocation. They start a catalogue of such assumption types, based on the assumptions discovered in their case studies. To discover these, their investigation consisted of manual inspection of the source code and developer interviews.

AA also proposes a followup that, to the best of our knowledge, has not yet been performed. It consists in codifying the assumptions such that these can be "used to semi-automatically identify assumptions in other aspect code" [10]. This would allow, on the one hand for implicit assumptions to be elicited from the source code, and on the other hand for explicit assumptions to be verified. For the latter, the ideal case would be "making fully automatic verification a feasible goal for at least some of the assumption categories" [10]. In this

section we show how GASR can be used to perform exactly this. We implement elicitation rules for a subset of the aspect assumptions and run them on two of the three case studies used in AA[6]. We consider that providing a complete set of rules would be a separate contribution and hence out of the scope of this work.

Concretely, we restrict ourselves to inter-aspect assumptions (Sect. 3.1.1 in [10]) and run the experiments on the HealthWatcher [27] and MobileMedia [28] systems. We implemented analysis rules for all assumptions that can be sufficiently formalized, or approximated by a heuristic. All rules were developed on a test-first basis and both the rules and the test cases are available online[7]. After running the analyses, the results were verified for correctness and completeness. This was achieved by manually inspecting both the source code as well as the full list of assumption instances published as additional material of the AA paper[8]. Our results confirm the assumption instances listed and, more importantly, provide new assumption instances that were overseen in AA. The latter clearly demonstrates the advantages of automatic aspectual source code reasoning, as provided by GASR.

Due to lack of space, we cannot fully document all the analyses we created for assumption identification. Instead we choose to focus here on interesting analyses: those that achieve fully automatic verification, reveal new assumption instances and show customizability.

A. Assumptions on concretisation of pointcuts

The first assumption we talk about here was already mentioned in Sect. II-C: an abstract pointcut that is concretized in a subclass and re-concretized in one of its subclasses. The assumption is that in such a case the aspect actually wishes to preserve existing behavior and hence should not override already concretised pointcuts. We can use GASR to perform fully automatic verification of this assumption, yielding a first step of the followup work proposed in the AA paper. The following logic rule will reveal violations of the assumption:

```
1 (defn pointcut-concretized-reconcretized
2  [?pointcut ?cpointcut ?rcpointcut]
3  (all
4    (pointcut-concretizedby ?pointcut ?cpointcut)
5    (pointcut-concretizedby ?cpointcut ?rcpointcut)))
```

In line 4 of the code above, we find a `?pointcut` that is concretised by a second `?cpointcut`, and in line 5 we find a `?rcpointcut` that concretises `?cpointcut`. Any solutions for this goal hence consist of a `?pointcut` that is concretised by `?cpointcut` and reconcretised by `?rcpointcut`.

We have queried both example case studies for matches of this rule and have found none. In other words there are no cases where this assumption has been violated. This is in accordance to the results published in AA.

B. Precedence assumptions

ASPECTJ provides for a mechanism to order the execution of advice when multiple advice apply at a given join point. It consists of precedence relations between different aspects and advice. This results in a domination order that determines the execution order of advice. The language contains implicit precedence rules that determine dominance between the aspects in an inheritance tree. Additionally, dominance between advice of the same aspect is determined by their order in the source code. The developer may also explicitly declare precedence between different aspects. One precedence assumption stated in AA is that implicit precedence rules between aspects are not modified by explicit precedence declarations. GASR can also be used to provide fully automatic verification of this assumption, as follows:

```
1 (defn overridden|imp|precedence [?asp1 ?asp2]
2  (all
3    (aspect|dominates-aspect ?asp2 ?asp1)
4    (aspect|implicitdominates-aspect+ ?asp1 ?asp2)))
```

Line 3 of the above rule provides bindings for domination relationships between aspects that have been explicitly declared, while line 4 succeeds for implicit domination relationships that are the opposite. The resulting bindings hence violate the aspect assumption. We have found none in the case studies, again in accordance to the results found in AA.

C. Inclusion assumptions of aspects

AA describes inclusion assumptions of aspects in general as "Some aspects require other aspects to be deployed to function correctly." This assumption cannot be unambiguously defined in a code rule. The paper however also identifies a specific variant: an aspect defines a marker interface, *i.e.,* an empty interface, and another aspect contains a declare parents statement that adds it as an implemented interface to a given class. The cases identified in the code studied for AA are actually a generalization of this: the interface sometimes is stand-alone, *i.e.,* defined in its own compilation unit. Moreover, aspects may refer to a sub-interface of this interface. The rules below successfully identify these assumption instances:

```
1(defn markerinterface [?interface]
2  (fresh [?member]
3    (interface ?interface)
4    (fails (type-member ?interface ?member))))

5(defn aspect-declareparents|markerinterface
6  [?aspect ?interface]
7  (fresh [?superinterface ?declare]
8    (markerinterface ?superinterface)
9    (iface-self|or|sub ?superinterface ?interface)
10   (declare|parents-parent|type ?declare ?interface)
11   (aspect-declare ?aspect ?declare)))
```

This code first defines a rule for a marker interface: an interface (line 3) that fails to have any members (line 4), *i.e.,* is a marker interface. This is then used in the assumption rule as a goal in line 8. Line 9 provides bindings for the interface and all its direct and indirect subinterfaces in `?interface`. As a result, line 10 succeeds on all declare parents statements of marker interfaces or their (in)direct subinterfaces. Line 11 reveals the `?aspect` that contains this declaration.

The above rules do not only identify the known instances of this assumption. More importantly, they also reveal three previously unidentified instances in the HealthWatcher case. Firstly, the ServletCommanding aspect refers to the CommandReceiver empty interface, which is stand-alone and

[6]The third system investigated in AA fails to compile due to an ASPECTJ internal compiler error and hence could not be analysed by us.

[7]Available at https://github.com/cderoove/damp.ekeko.aspectj

[8]Available at http://www.steffen-zschaler.de/publications/rivar_data/

specifically designed for aspects to use as a marker interface, as revealed by its comments. Secondly, the UpdateStateObserver aspects refers to the Observer interface contained in the ObserverProtocol aspect. This nested interface was also created specifically for other aspects to mark, as indicated by its comments. Thirdly, analogous to the previous instance, UpdateStateObserver also refers to the Subject interface contained in the ObserverProtocol aspect, also created for this. It is unclear why these are not present in the AA raw data as they are indubitably assumption instances.

D. Mutual Exclusion Assumptions

AA states that "aspects may also be mutually exclusive", *i.e.,* of the mutually exclusive set only one aspect may be deployed. Again, this assumption cannot be unambiguously defined in a code rule. We can infer some heuristics that can however give possible cases for such a mutual exclusion. Based on the conjecture that mutually exclusive aspects may provide different implementations for the same feature and hence act on the same parts of the software, we present two such heuristics here: the same pointcut name and the same join point shadows. Note that we do not claim that this conjecture and the heuristic is particularly efficient, nor even valid. These only serve as an illustration of the use of GASR.

1) Same Pointcut Name: For this heuristic we assume that the name of the pointcuts convey their semantics and hence if two aspects use pointcuts with the same name they may implement the same feature. The following rule reveals such aspects:

```
1 (defn same|pointcutname-aspect1-aspect2
2  [?name ?aspect1 ?aspect2]
3    (fresh [?pc1 ?pc2]
4      (differs ?aspect1 ?aspect2)
5      (aspect-pointcutdefinition ?aspect1 ?pc1)
6      (aspect-pointcutdefinition ?aspect2 ?pc2)
7      (pointcutdefinition-name ?pc1 ?name)
8      (pointcutdefinition-name ?pc2 ?name)))
```

The code of the rule is straightforward, obtaining pointcut definitions of two different aspects where the name of the pointcut is the same. For the HealthWatcher case this rule only reveals two cases where an abstract pointcut is concretized. In MobileMedia however 146 cases are detected, defying manual analysis of each case. It is immediately apparent that a small subset of pointcut names are present a sizeable amount of times: "handleCommandAction", "initMenu" and "constructor". This is as many aspects are used to implement a command pattern and match on these pointcuts to realize the pattern. Using GASR we can eliminate these matches from the rule by amending extra conditions to the query, as below:

```
1(ekeko [?name ?as1 ?as2]
2  (all
3    (same|pointcutname-aspect1-aspect2 ?name ?as1 ?as2)
4    (differs ?name "handleCommandAction")
5    (differs ?name "constructor")
6    (differs ?name "initMenu")))
```

Running this query returns in eleven different matches, which is a number that allows for manual analysis. This actually reveals six cases of copy-paste reuse of a pointcut: "createMediaData", "getMediaController", "goToPreviousScreen", "initForm", "appendMedias" and "startApp". The two remaining pointcut names: "resetMediaData" and "showImage" do not reveal mutual exclusion of aspects.

The use of this heuristic did not reveal assumption instances but is nonetheless valuable. This is as its use in the MobileMedia case study illustrates the advantage of a general-purpose code reasoner to adapt code queries to the actual case being studied, in this case filtering out a high number of false negatives. This resulted in the discovery of six cases of copy-paste reuse, arguably not a good characteristic of the code.

2) Same join point shadows: Using the same pointcut names is not the only possible indication of implementing the same features. This second heuristic considers the join point shadows of two aspects. If two different aspects have the same collection of shadows, they may implement the same feature. This can be detected using the code below.

```
1 (defn  sameshadows|aspect1-aspect2
2  [?aspect1 ?aspect2]
3  (fresh [?shadows1 ?shadows2]
4    (aspect ?aspect1) (aspect ?aspect2)
5    (differs?aspect1 ?aspect2)
6    (findall ?shadow1
7      (aspect-shadow ?aspect1 ?shadow1) ?shadows1)
8    (findall ?shadow2
9      (aspect-shadow ?aspect2 ?shadow2) ?shadows2)
10   (differs ?shadows1 []) (differs ?shadows2 [])
11   (same-elements ?shadows1 ?shadows2)))
```

Notable here are lines 6 and 8: all shadows of both aspects are gathered in the collections `?shadows1` and `?shadows2`, respectively. Line 10 ensures that the collections are not empty, to exclude aspects without advice. Line 11 verifies that the collections have the same elements, *i.e.,* are the same.

For the HealthWatcher case two matches are found: HWTransactionExceptionHandler and HWDistributionExceptionHandler. Both aspects perform complementary exception handling: one for transaction exceptions and one for RMI exceptions. In MobileMedia nine different matches are found, of which one is a mutual exclusion case: OneAlternativeFeature and TwoAlternativeFeatures. Both add an exit command to the same menu and hence are mutually exclusive. This case is not mentioned in the AA paper but is present in the raw data.

E. Assumptions on the use of Inter-Type Declarations

One kind of purpose for inter-type declarations is to provide additional public features that are packaged in the aspect. In these cases these methods "are often not used from the declaring aspect" [10], so the assumption is that other code will call these aspects at some points. GASR accomplishes automatic verification of this assumption by reasoning over the results of our extended soot analysis, discussed in Sect. IV-B. The code is as follows:

```
1(defn intertypemethod|unused [?itmethod]
2  (fresh [?sootmethod ?caller]
3    (intertype|method ?itmethod)
4    (fails (all
5      (intertype|method-soot|method
6          ?itmethod ?sootmethod)
7      (soot|method|callee-soot|method|caller
8          ?sootmethod ?caller)))))
```

In the code above, line 3 provides bindings for methods that are inter-type declarations in `?itmethod`. Lines 5 and 6

provide the bridge between the method and the corresponding soot method, and lines 7 and 8 find methods that call that soot method, binding these to `?caller`. The goal in line 4 fails if the goals in lines 5–8 succeed, *i.e.,* if no bindings can be found for `?caller`. As a result the rule succeeds for inter-type declarations that are not called.

In the case studies we have found one violation of this assumption in HealthWatcher: Command.isExecutable() in the CommandProtocol aspect is a default implementation for the abstract method declared in the Command class. Yet this method is never referenced at all. A record of this violation of the assumption is as it is not present in the raw data of AA. Hence this is again a new discovery made thanks to GASR.

F. Conclusion

In this section we have illustrated the usefulness of GASR by implementing detection of aspect assumptions, as originally envisioned by Zschaler and Rashid [10], which we name AA for brevity. We have implemented logic rules for the detection of inter-aspect assumptions, effectively achieving some of the future work laid out in AA. We have run these rules on two of the three systems used in AA for discovering the aspect assumptions. The results of these GASR queries were verified for correctness and completeness by manually inspecting both the source code as well as by cross-checking with the full published list of assumption instances.

All the assumption instances that are present in the AA paper or in the raw data were detected using GASR. More important though are the three following results: First, we achieved fully automatic verification of assumption instances in Section V-A and V-B. Second, we also detected three previously unknown assumption instances in Section V-C and one in Section V-E. Third, we have shown in Section V-D1 how the general-purpose nature of GASR enables the tailoring of an existing rule to the software under study. Thanks to this, we incidentally found six cases of copy-paste reuse of pointcuts.

VI. Threats to Validity

We consider GASR a reasonable baseline for a general-purpose source code analysis tool for aspect-oriented programming. However we can not and do not claim that it is suitable for all possible kinds of reasoning over aspectual source code.

Considering the above restrictions, the first threat to validity is the fact that we only performed the experiments on two concrete cases. Nonetheless, the rules were developed independently of the case studies, on a test-first basis, and should hence perform equally well on other case studies. Secondly, we have only shown here that GASR works for rules from the work on Aspect Assumptions [10]. As highlighted in Section III, there is a large amount of work that performs analysis of aspectual code and we do not validate that GASR is as effective for those cases. By providing a comprehensive library of predicates, discussed in Section IV-B, we do however provide a large number of basic building blocks that can be used to build these analyses using GASR. It remains to be shown whether the library is extensive enough. If not, it may need to be extended.

GASR is a source code analysis tool that works, as-is, on ASPECTJ source code only. It has however been argued before, *e.g.,* by Zschaler and Rashid [10], that ASPECTJ is probably the most used aspect-oriented language. As a result, GASR can be used to analyse a large amount of aspectual source code. Moreover, the predicate library is relatively language-agnostic as it works in terms of the aspect-oriented concepts. We are confident that if the library is adapted to work on other languages, the majority of analyses built using GASR will be straightforwardly reusable. We have demonstrated similar results previously in earlier work on language-independent source code analysis [29]. Hence, in our opinion, GASR truly is general-purpose.

VII. Conclusion and Future Work

There is a need for source code analysis of aspect-oriented source code that is demonstrated by the multiple tracks of research performing such analysis. On the one hand, existing analyses need to be extended to take into account the aspect-oriented nature of the software, and on the other hand this nature gives rise to new kinds of analyses being required. Yet, to the best of our knowledge, all of this work has been ad-hoc and limited in scope to the specific analysis at hand. As a result there has been a significant amount of duplicate work and it is unclear whether any analyses may be customized to the software being analysed, *e.g.,* as we perform in Section V-D1.

We state that what is required is a general-purpose aspectual source code analysis tool, such that duplicate work building analyses may be avoided and that existing analyses may be customized to the task at hand. To the best of our knowledge no such work has yet been published.

To address this need, we have implemented GASR: a source code analysis tool in the tradition of logic querying. GASR is a General-purpose Aspectual Source code Reasoner whose analyses may be customized relatively straightforwardly, as illustrated in Section V-D1. In this paper we presented GASR, have shown illustrative predicates for the reification of structural and behavioral relations, and discussed their implementation.

Following this, we performed source code analysis on two representative pieces of aspect-oriented software. We detected a subset of inter-aspect assumptions that were previously identified by Zschaler and Rashid [10] by manual inspection of the same software. Our automated analysis effectively consists in realizing part of the future work outlined in that text: allowing detection of assumptions and fully automatic verification that some assumptions are not being violated. We detected the same assumption instances as Zschaler and Rashid. More importantly, we also found assumption instances that were overlooked by these authors.

There are multiple avenues for possible future work. Firstly, we contend that the current state of GASR is a reasonable baseline for performing aspect-oriented source code analysis but do not assert that it is sufficient for all kinds of analyses. More experiments, implementing different analyses and executing them on multiple case studies may reveal areas where GASR is lacking. Secondly, the assumptions of Zschaler and Rashid [10] which we did not implement yet are an avenue for further work. Some of these however require the source code to be annotated somehow with the intent of the developer. This would require some formal notation for these intentions and GASR to be extended to reason over these annotations.

Thirdly, GASR can be seen as base infrastructure on which new and more advanced analyses can be built. The possibilities are vast, obviously. Our personal preferences are for analyses that can extract design-level documents [30] and that provide information on whether there exist any dependencies and interactions between aspects [31].

ACKNOWLEDGMENTS

Johan Fabry is partially funded by FONDECYT project number 1130253, Coen De Roover is funded by the *Cha-Q* SBO project sponsored by the "Flemish agency for Innovation by Science and Technology" (IWT Vlaanderen). Thanks to Romain Robbes for feedback on draft versions of this text.

REFERENCES

[1] L. Ye and K. De Volder, "Tool support for understanding and diagnosing pointcut expressions," in *Proceedings of the 7th international conference on Aspect-oriented software development*, ser. AOSD '08. New York, NY, USA: ACM, 2008, pp. 144–155.

[2] R. Khatchadourian, P. Greenwood, A. Rashid, and G. Xu, "Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software," in *24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*, 2009, pp. 575–579.

[3] C. Koppen and M. Stoerzer, "Pcdiff: Attacking the fragile pointcut problem," in *European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[4] A. Kellens, K. Mens, J. Brichau, and K. Gybels, "Managing the evolution of aspect-oriented software with model-based pointcuts," in *European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, no. 4067, 2006, pp. 501–525.

[5] D. Zhang, E. Duala-Ekoko, and L. Hendren, "Impact analysis and visualization toolkit for static crosscutting in aspectj," in *International Conference on Program Comprehension (ICPC)*, 2009.

[6] H. Shen, S. Zhang, J. Zhao, J. Fang, and S. Yao, "Xfindbugs: extended findbugs for aspectj," in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE08)*, 2008, pp. 70–76.

[7] G. Xu and A. Rountev, "Ajana: a general framework for source-code-level interprocedural dataflow analysis of aspectj software," in *Proceedings of the 7th international conference on Aspect-oriented Software Development (AOSD08)*, ser. AOSD '08, 2008, pp. 36–47.

[8] A. Colyer, A. Clement, G. Harley, and M. Webster, *Eclipse AspectJ: aspect-oriented programming with AspectJ and the Eclipse AspectJ development tools*. Addison-Wesley Professional, 2004.

[9] J. Fabry, A. Kellens, and S. Ducasse, "AspectMaps: A scalable visualization of join point shadows," in *Proceedings of 19th IEEE International Conference on Program Comprehension (ICPC2011)*. IEEE, Jul 2011, pp. 121–130.

[10] S. Zschaler and A. Rashid, "Aspect assumptions: a retrospective study of aspectj developers' assumptions about aspect usage," in *Proceedings of the tenth international conference on Aspect-oriented software development*, ser. AOSD '11. New York, NY, USA: ACM, 2011, pp. 93–104.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An overview of AspectJ," in *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, ser. Lecture Notes in Computer Science, J. L. Knudsen, Ed., no. 2072. Budapest, Hungary: Springer-Verlag, Jun. 2001, pp. 327–353.

[12] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein, "Multijava: modular open classes and symmetric multiple dispatch for java," in *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '00. New York, NY, USA: ACM, 2000, pp. 130–145.

[13] J. Xu, H. Rajan, and K. Sullivan, "Understanding aspects via implicit invocation," in *Proceedings. 19th International Conference on Automated Software Engineering (ASE)*, 2004, pp. 332–335.

[14] S. Hanenberg and A. Schmidmeier, "Idioms for building software frameworks in aspectj," in *Proceedings of the workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) at AOSD 2003*, 2003, p. 55.

[15] R. Laddad, *AspectJ in action*, 2nd ed. Manning Publications, 2009.

[16] J. Fabry, A. Kellens, S. Denier, and S. Ducasse, "AspectMaps: Extending Moose to visualize AOP software," *Science of Computer Programming*, 2013, To appear. http://dx.doi.org/10.1016/j.scico.2012.02.007.

[17] E. Hajiyev, M. Verbaere, and O. de Moor, "CodeQuest: Scalable source code queries with Datalog," in *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP06)*, ser. Lecture Notes in Computer Science, vol. 4067, 2006, pp. 2–27.

[18] M. Martin, B. Livshits, and M. Lam, "Finding application errors and security flaws using PQL: a program query language," in *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA05)*, 2005, pp. 365–383.

[19] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers, "The SOUL tool suite for querying programs in symbiosis with eclipse," in *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ11)*, 2011.

[20] W. E. Byrd, "Relational programming in minikanren: Techniques, applications, and implementations," Ph.D. dissertation, Indiana University, August 2009.

[21] A. K. Carlos Noguera, Coen De Roover and V. Jonckers, "Program querying with a SOUL: the barista tool suite," in *Proceedings of the 27th IEEE International Conference on Software Maintenance, Tool Demo Track (ICSM11)*, 2011.

[22] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, Oct. 2011.

[23] R. Stevens, C. De Roover, C. Noguera, and V. Jonckers, "A history querying tool and its application to detect multi-version refactorings," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013)*, 2013.

[24] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI06)*, 2006.

[25] O. de Moor, D. Lacey, and E. V. Wyk, "Universal regular path queries," *Higher-order and Symbolic Computation*, vol. 16, no. 1-2, pp. 15–35, 2003.

[26] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu, "Parametric regular path queries." in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI04)*, 2004, pp. 219–230.

[27] S. Soares, P. Borba, and E. Laureano, "Distribution and persistence as aspects," *Software: Practice and Experience*, vol. 36, no. 7, pp. 711–759, 2006.

[28] E. Figueiredo, I. Galvao, S. Khan, A. Garcia, C. Sant'Anna, A. Pimentel, A. Medeiros, L. Fernandes, T. Batista, R. Ribeiro, P. van den Broek, M. Aksit, S. Zschaler, and A. Moreira, "Detecting architecture instabilities with concern traces: An exploratory study," in *Proceedings of 8th Joint Working IEEE/IFIP Conference on Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009.*, 2009, pp. 261–264.

[29] J. Fabry and T. Mens, "Language-independent detection of object-oriented design patterns," *Science of Computer Programming*, vol. 30, no. 1-2, pp. 21–33, April-July 2004.

[30] J. Fabry, A. Zambrano, and S. Gordillo, "Expressing aspectual interactions in design: Experiences in the slot machine domain," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, J. Whittle, T. Clark, and T. Kühne, Eds. Springer Berlin / Heidelberg, 2011, vol. 6981, pp. 93–107.

[31] R. Chitchyan, J. Fabry, S. Katz, and A. Rensink, "Editorial for special section on dependencies and interactions with aspects," *Transactions on Aspect-Oriented Software Development*, vol. LNCS 5490, pp. 133–134, 2009.