AspectJ Code Analysis and Verification with GASR[☆]

Johan Fabry[a,*], Coen De Roover[b], Carlos Noguera[b], Steffen Zschaler[c], Awais Rashid[d], Viviane Jonckers[b]

[a] *PLEIAD Laboratory, Computer Science Department (DCC), University of Chile*
[b] *Software Languages Lab, Vrije Universiteit Brussel*
[c] *Department of Informatics, King's College London*
[d] *Computer Science Department, Lancaster University*

Abstract

Aspect-oriented programming languages extend existing languages with new features for supporting modularization of crosscutting concerns. These features however make existing source code analysis tools unable to reason over this code. Consequently, all code analysis efforts of aspect-oriented code that we are aware of have either built limited analysis tools or were performed manually. Given the significant complexity of building them or manual analysis, a lot of duplication of effort could have been avoided by using a general-purpose tool. To address this, in this paper we present GASR: a source code analysis tool that reasons over ASPECTJ source code, which may contain metadata in the form of annotations. GASR provides multiple kinds of analyses that are general enough such that they are reusable, tailorable and can reason over annotations. We demonstrate the use of GASR in two ways: we first automate the recognition of previously identified aspectual source code assumptions. Second, we turn implicit assumptions into explicit assumptions through annotations and automate their verification. In both uses GASR performs detection and verification of aspect assumptions on two well-known case studies that were manually investigated in earlier work. GASR finds already known aspect assumptions and adds instances that had been previously overlooked.

Keywords: Aspect Oriented Programming, Source Code Analysis, Logic Program Querying, Aspectual Assumptions

[☆] This is an extended edition of the article *Aspectual Source Code Analysis with GASR* by Fabry, De Roover and Jonckers, presented at the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'13)[1].

[*] Corresponding author

Email addresses: jfabry@dcc.uchile.cl (Johan Fabry), cderoove@vub.ac.be (Coen De Roover), cnoguera@soft.vub.ac.be (Carlos Noguera), szschaler@acm.org (Steffen Zschaler), marash@comp.lancs.ac.uk (Awais Rashid), vejoncke@soft.vub.ac.be (Viviane Jonckers)

1. 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 kind of module that encapsulate, in addition to their behavior, when this behavior needs to be invoked; that is, they 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 amongst themselves and with the non-aspectual language features. An example of the latter interaction is that an aspect may change the class hierarchy, an example of the former interaction is pointcuts that are slightly or subtly incorrect and hence fail to match or match wrongly. As a result, many possible issues in aspectual code have been separately identified and source code analysis tools built to address them [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13].

Yet, to the best of our knowledge, all of these kinds of analyses have been made on an ad-hoc or limited basis and were customized specifically to the analysis task being performed. This is far from trivial work, given the complexity of building a source code reasoner, or adding the novel aspectual language fatures to an existing reasoner. As a result there has been a considerable duplication in development effort and time wasted. Moreover the resulting code reasoners are not customizable to the actual software under analysis, *e.g.,* to automatically remove one kind of known false positives from the results, or to augment or weaken the analysis being performed.

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. Moreover, it should take into account that source code may be augmented by metadata in the form of annotations, *e.g.,* to explicitly mark methods as overriders as in the Java `@Override` annotation. Such metadata may also be used in aspectual code, hence a source code reasoner for aspects should also be aware of annotations. To address the need for such a source code reasoner, we built GASR and we present it in this paper.

The novelty of this work lies in that GASR provides multiple kinds of source-

code analysis over aspect-oriented software that are general enough such that they can be used for multiple types of analyses, can reason over metadata in annotations, and moreover are adaptable such that they can be tailored to the task at hand.

More specifically, 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 for ASPECTJ-like languages [14] and discusses its implementation along with its library of logical predicates that can be used to reason both about structure as well as behavior of aspectual source code.
- It shows how GASR can be used to automatically verify a subset of previously published inter-aspect assumptions [15] that are implicit in the code.
- It shows how an additional set of implicit aspect assumptions, from the same work, can be made explicit through annotations, and how GASR can verify these explicit assumptions.

Note that the last two items in the list above correspond to effectively implementing part of the future work identified in the text on aspectual assumptions [15]. The term *aspectual assumptions* is used in [15] to identify assumptions made by aspect developers about the context in which their aspect will be woven. For example, aspect developers might assume that certain other aspects will also be woven, that a pair of pointcuts identifies joinpoints that are in a particular control- or data-flow relationship, or that certain abstract pointcuts will be instantiated to identify points in the system execution with a certain meaning. Current aspect languages do not allow these assumptions to be made explicit, so that they cannot be easily captured let alone validated whenever an aspect's usage context changes. This leads to a number of fragility issues with aspect definitions, making their use and reuse difficult.

GASR was originally presented in previous work [1]. This journal paper extends the conference publication in multiple ways. It firstly extends the analysis of related work to more clearly show where existing tools fall short when compared to GASR and provides a more thorough discussion section. Secondly it expands GASR to include analysis of metadata in the form of annotations, expanding the reasoning capabilities of the underlying engine. Thirdly it contains additional evaluation work. This through the more complete application to the scenarios found in the aspectual assumptions paper, which required definitions of explicit assumptions as annotations, specifying the reasoning rules and inserting the annotations in the source code.

The structure of this text is as follows: first we provide the problem statement of the paper, arguing for the need for GASR. In Section 3 we give an overview of related work, showing that existing analyses have been ad-hoc or limited. We then present GASR in Section 4, discussing its implementation and

a selection of its library of logical predicates. This is followed by an illustration of the usefulness of GASR: realizing detection of implicit aspectual assumptions in Section 5, as previously identified in [15], and detection of explicit aspectual assumptions in Section 6, extending the same work. The paper then provides a discussion in Section 7 before presenting conclusions and avenues for possible future work.

2. 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.

2.1. 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 [14], arguably the best-known and most-used aspect language. ASPECTJ is an extension of Java that introduces aspect features using a specific syntax. In AspectJ, 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 [16].

2.2. 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(* *(..)) { 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.

2.3. Problem: A New Reasoning Need

If we consider aspects as simply a means to achieve behavior subject to implicit invocation with implicit announcement [17], 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

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

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 [15], *i.e.,* the re-concretization in Grandchild.

As a result, in addition to the classic example of Sect. 2.2, 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 [15], which we will discuss in more detail in Section 5. Second is the problem of pointcuts that are slightly or subtly incorrect [2], 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 [6, 7], 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.

Also, multiple aspectual design patterns have been presented [18, 19], yet no mining of these patterns with a source code reasoner has been documented. A well-known example is the Wormhole [19]: 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.

Lastly, in addition to executable code, source code can contain annotations that give meta-information, which may also be considered as some form of contract on the source. An example of this is the Java @Override annotation, placed on methods that are meant to override other methods defined higher up in the hierarchy. The compiler checks if the overriden method actually exists and will raise a warning when this is not the case, *e.g.,* if there is a typo in the method name or the argument types do not match. Such explicit annotation of contracts, together with the verification of these contracts may also be beneficial for aspect-oriented code, as has been mentioned in previous work [15] and will be discussed in Section 6. Yet no aspect-oriented code reasoner exists that takes annotations into account.

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, can reason over metadata in annotations, 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. 2.2, or detect incorrect use of the Wormhole pattern.

3. 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 or limited analyses made, and indirectly related is work on code comprehension of aspectual source code, source code reasoning with annotation support and aspect verification.

Getting pointcuts correct can be a hard task [2], 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 [2], 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. Wloka et.al. [3] presented a meta-model for pointcut representation and an impact analysis for pointcuts so that source code refactoring is aspect-aware. As a result, refactorings that impact shadow matches also include refactoring proposals for the pointcut, such that these remain valid. In the same line we find the work on pointcut rejuvenation and Fraglight [4, 5]. 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. Custom analyses are developed that suggests changes to pointcuts when needed, either considering the aspect developer [4] or the base code developer [5]. Earlier work in this area [6, 7] also used custom analyses.

Yet reasoning about aspectual source code is not limited to pointcuts only. For example, ITDVisualizer [8] 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 [9] 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]. Again all of the above tools use a custom reasoner to provide an analysis that is limited to producing these specific results.

Zhao proposes a change impact analysis for aspect-oriented software [10] by establishing control and data dependencies. The work effectively extends object-oriented change analysis to provide support for aspect-oriented code. Last but not least, the work on Ajana [11] focuses on dataflow analysis for AspectJ code. More specifically, Ajana is a framework for source-code-level interprocedural dataflow analysis, using a control- and data-flow program representation.

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

It provides an object effect analysis and a dependency analysis based on this representation. Being focused on dataflow analysis, it however does not provide functionality, *e.g.,* to fully reason over pointcuts or over intertype declarations.

	[2]	[3]	[4, 5]	[6, 7]	[8]	[9]	[10, 11]	GASR
Pointcut Shadows	+	+	+	+	-	+	-	+
Pointcut Structure	+	+	-	-	-	+	-	+
Object Structure	+	+	+	-	+	+	-	+
Aspect Structure	-	+	+	-	+	+	-	+
Intertype Structure	-	-	-	-	+	+	-	+
Control Flow	-	+	-	-	-	+	+	+
Data Flow	-	-	-	-	-	-	+	+
Annotations	-	-	-	-	-	-	-	+
Customize/Extend	-	-	-	-	-	-	-	+

Table 1: Aspectual source code reasoners: does the work reason about ... and can a user extend or customize the reasoning. In contrast to GASR, none of previous tools cover all facets and moreover none is extensible or customizable.

In Table 1, we give an overview of what parts of the source code is analyzed by the different reasoners created for this related work. Notably, none of the reasoners cover all parts of aspect-oriented source code, and all are restricted to the specific analyses required for the task at hand. They cannot be extended with new analyses or the existing analyses cannot be customised by the user. In contrast, GASR covers all parts of aspect-oriented source code and is extensible and customizable.

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

The AspectJ Development Toolkit (AJDT) [12] 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 [13, 20]. 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 [13]. 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 run-time 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.

To the best of our knowledge, no aspectual source code reasoner exists that also takes annotations into account. Existing work on source code analysis with annotations are purely for non-aspectual code, with different focuses: explicating contracts for the design-by contracts approach as annotations [21], pluggable type systems that use annotations to express type qualifiers [22] or take annotations into account when checking [23], keeping annotations and source code consistent when either is evolved [24].

In Section 6 we show how explicit annotations to AspectJ source code can be used to validate developer assumptions automatically. This is closely related to the verification of aspect-oriented software. One system that has previously explored aspect verification is MAVEN [25], which provides modular verification of LTL properties of aspects. To provide practical applicability, MAVEN has later been extended with an ability to derive LTL properties to be verified from English-language template descriptions [26]. This verification is more powerful than what we are presenting here, as the LTL approach naturally supports relating multiple points on an execution path and our proposed solution currently only relates two points on an execution path. However in MAVEN "aspects are specified directly as state machines" [25] in a language specific to MAVEN. As a consequence, MAVEN should not be considered as a source code reasoner but rather as a model checker, and a means to convert source code to a MAVEN model, to the best of our knowledge, does not exist.

4. 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?*". Gasr specifically targets programs implemented in AspectJ, arguably the most appropriate target language for an aspectual source code reasoning infrastructure. We discuss this choice in more detail in Section 7.

Users' 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 [27], PQL [28] and Soul [29].

GASR owes its query language to the CORE.LOGIC[3] port to Clojure of MINIKANREN [30], and its IDE integration to the EKEKO [31] Eclipse plugin. The EKEKO plugin enables launching logic program queries from within Eclipse. These queries are passed for evaluation to the logic engine of CORE.LOGIC, which returns a collection of source code elements from the queried program that satisfy all of the query's conditions. The EKEKO plugin subsequently presents these elements to the user for inspection. Section 4.1 introduces the query syntax and their evaluation.

GASR itself provides an extensive library of logic predicates to be used within queries about AspectJ programs. These predicates expose information from the AspectJ compiler about the state of the queried program. The source code elements returned for a query are in fact instances of various org.aspectj.weaver classes, more specifically those that were used during the most recent compilation of the AspectJ program in the Eclipse workspace. As such, solutions to a query always correspond to the current state of the queried program. Section 4.2 details the GASR predicate library and their implementation.

4.1. 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 succeed. The `differs/2` goal on line 2 imposes a disequality

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

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.

4.2. The Predicate Library of GASR

To enable querying ASPECTJ programs, GASR provides 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 2.2:

```
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 2.3:

```
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 2 and Table 3 list representative predicates that reify structural resp. behavioral relations between AspectJ source code entities. We refer to the online documentation[4] for an overview of the complete predicate library. The remainder of this section discusses the highlights of its implementation.

4.2.1. Predicates Reifying Structural Relations

The predicates listed in Table 2 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 populated by the AspectJ weaver after each compilation.

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 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

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

Predicate	Reified Relation Of
`(type ?type)`	All types known to the weaver (*i.e.*, aspects, classes, interfaces, enums, *etc*).
`(type-declaredsuper ?type ?super)`	A type and its direct declared superclass or -aspect.
`(type-declaredinterface ?type ?interface)`	A type and one of the interfaces it declares to be implementing or extending directly.
`(type-super+ ?type ?super)`	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)`	A type and one of its declared methods.
`(type-method+ ?type ?method)`	A type and one of its declared or inherited methods. Does not include intertype declaration methods.
`(aspect ?asp)`	All known aspects. Subrelation of `type/2`.
`(aspect-pointcutdefinition ?asp ?pcdef)`	An aspect and one of its declared pointcut definitions.
`(aspect-advice ?asp ?adv)`	An aspect and one of its declared advice.
`(aspect-intertype ?asp ?intertype)`	An aspect and one of its intertype member declarations.
`(aspect-declare ?asp ?declare)`	An aspect and one of its `declare` declarations (*e.g.*, parents, precedence).
`(pointcutdefinition-pointcut ?pcdef ?pc)`	A non-abstract pointcut definition and its pointcut.
`(pointcutdefinition-name ?pcdef ?name)`	A pointcut definition and its name.
`(pointcutdefinition|abstract ?pcdef)`	Abstract pointcut definitions. Sub-relation of `pointcutdefinition/1`.
`(advice|before ?adv)`	`before` advices. Sub-relation of `advice/1`.
`(advice-pointcut ?adv ?pointcut)`	An advice and its pointcut. The latter either an anonymous pointcut, or a pointcut definition.
`(advice-pointcutdefinition ?adv ?pcdef)`	An advice and the concrete pointcutdefinition its name resolves to (*i.e.*, overrides of possibly abstract pointcutdefinitions are taken into account).
`(advice-shadow ?adv ?shadow)`	An advice and one of its join point shadows.
`(shadow-enclosing ?shadow ?enclosing)`	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)`	A shadow and its first enclosing type entity (*e.g.*, aspect, class, enum).
`(intertype-member-target ?itd ?mem ?type)`	An intertype declaration, the member (*i.e.*, field, method or constructor) it declares, and the type to which this member is added.
`(declare|parents ?dec)`	`declare parents`. Subrelation of `declare/1`.
`(declare|parents-target-parent ?dec ?target ?parent)`	`declare parents` with one of the target types matching its pattern and the corresponding supertype.
`(declare|precedence ?dec)`	`declare precedence`. Subrelation of `declare/1`.
`(aspect|dominates-aspect ?dasp ?sasp)`	Actual domination relations between aspects that result from `declare precedence` declarations.

Table 2: Representative predicates concerning structure.

on line 5 will therefore successively unify `?pcdef` with each of the elements of the returned collection of ResolvedPointcutDefinition instances.

4.2.2. Predicates for Annotations

In addition to reasoning about the code, GASR can also take into account the metadata placed in the source in the form of Java annotations. The type of such

annotations is codified through an annotation type declaration that essentially specfies a number of key-value attributes. Such annotations are also reified in GASR, which allows for them to be queried and their respective key-value pairs to be obtained.

For example, consider adding explicit labels to source code entities through a `@Label` annotation (which we will use in Section 6.2). First a type needs to be created, let's say we use the fully qualified name `damp.ekeko.aspectj.annotations.Label`, and it specifies that the label takes one argument: a string. Source code can then be queried for all these annotations by using the predicate below, and binding `?atn` to the fully qualified name `damp.ekeko.aspectj.annotations.Label` in the query.

```
1(defn type-annotation-annotation|type|name [?type ?ann ?atn]
2  (fresh [?at]
3    (type-annotation ?type ?ann)
4    (annotation-annotationtype ?ann ?at)
5    (type-name ?at ?atn)))
```

The goal on line 3 successively binds `?ann` with each annotation of a type `?type`, in turn successively bound to each type. In line 4 `?at` is bound to the annotation type declaration of the annotation, and in line 5 `?atn` unifies with the name of this type.

To be able to reason about the contents of the annotation, the following predicate exposes the key and value pairs present in an annotation. It relies on the previous predicate to obtain an annotation of which its containing type and annotation type name are bound (in line 4) to the logic variables given as argument. For these annotations `?ann`, line 5 successively binds the different `?key` and `?value` combinations through backtracking.

```
1(defn type-annotation|key-annotation|value-annotation|type|name
2  [?type ?key ?value ?atn]
3  (fresh [?ann]
4    (type-annotation-annotation|type|name ?type ?ann ?atn)
5    (annotation-key-value ?ann ?key ?value)))
```

The above predicate can be used in the same way, binding `?atn` to the fully qualified name `damp.ekeko.aspectj.annotations.Label` in the query. If such queries are frequently used it is however bothersome to need to specify the fully qualified name every time. One option for a programmer would be to add a custom predicate to the system to simplify things as follows:

```
1(defn labeled|type-label|val [?type ?val]
2  (fresh [?key]
3    (type-annotation|key-annotation|value-annotation|type|name ?type ?key ?val
4       "damp.ekeko.aspectj.annotations.Label")))
```

This predicate successively binds types to `?type` and successively the values of the labels present to `?val`. To do this, it uses the previous predicate to bind the label annotations in lines 3 and 4. From these annotations only the values

are bound, because since labels only have one key-value pair there is no need to consider the key.

An alternative to the use of fully qualified names for types would be to use type patterns, as present in AspectJ. This not only shortens the specification

Predicate	Reified Relation Of
`(advice|reads-field ?advice ?field)`	An advice and one of the fields it reads from.
`(advice|writes-field ?advice ?field)`	An advice and one of the fields it writes to.
`(advice-reachable|advice ?advice1 ?advice2)`	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.
`(behavior-reachable|behavior ?beh1 ?beh2)`	An advice, method or constructor and another advice, method or constructor that have the same relation as above.
`(field-soot|field ?field ?soot)`	A field and the Soot field that represents its implementation.
`(advice-soot|method ?advice ?soot)`	An advice and the Soot method that represents its implementation.
`(intertype|method-soot|method ?itmethod ?soot)`	A method declared by an intertype declaration and the Soot method that represents its implementation.
`(soot|method-soot|unit ?method ?unit)`	A Soot method and one of the units in its body. These correspond to instructions in Soot's Jimple intermediate representation [32] of the woven program.
`(soot|unit|reads-soot|valuebox ?unit ?val)`	A Soot unit and one of the values (*e.g.*, parameters, field references, expressions...) it reads from.
`(soot|unit|writes-soot|valuebox ?unit ?val)`	A Soot unit and one of the values it writes to.
`(icfg|main-start ?icfg ?icfg|start)`	The inter-procedural control flow graph of the woven program and its starting node.
`(icfgnode-unit ?node ?unit)`	A node of the inter-procedural control flow graph and a Soot unit.
`(icfgnode-method ?node ?method)`	A node of the inter-procedural control flow graph and the Soot method in which it resides.
`(icfgnode-stack ?node ?stack)`	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 [`$v_1 \ldots v_n$`] `$q_1 \ldots q_n$`)`	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 [33]: `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 3: Representative predicates concerning behavior.

of types, but also gives more flexibility to the programmer when specifying type names (as will be shown in Section 6.1).

We have included type pattern matching in GASR by simply hooking into the ASPECTJ compiler's pattern matching logic, and extending it with the logic for an extra wildcard: =. As a result, the semantics of type patterns we use is an extension of the ASPECTJ semantics:

- The * wildcard on its own matches all types.
- The * wildcard matches zero or more characters except for the dot.
- The .. wildcard matches any sequence of characters that start and end with a dot.
- The + wildcard matches all subtypes of a type (or a collection of types), including the type itself. It needs to immediately follow a type name pattern.
- The = wildcard matches all subtypes of a type (or a collection of types), except for the type itself. It needs to immediately follow a type name pattern.

Considering the Label example, using wildcards would allow us to specify `*..Label` instead of `damp.ekeko.aspectj.annotations.Label` in queries.

4.2.3. Predicates Reifying Behavioral Relations

The predicates listed in Table 3 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 3 reify behavioral relations between elements that stem from the weaver's domain model. These can be combined with the structural predicates of Table 2. 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 [32] analysis framework (third row in Table 3) and predicates that link both sources of information together (second row in Table 3). 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 [32] 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 3). 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. [34]. 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 3, `path/n` is of special interest as it embodies the implementation of parametric regular path expressions [35, 36] in EKEKO (which we have applied in earlier work to query the history of versioned software [33]). Regular path expressions 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`:

```
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 2.3. These are characterized by an execution path on which the wormholed field is written to inbetween the entry and exit advice.

5. Detecting Implicit 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. This effectively extends the "Aspect Assumptions" work of Zschaler and Rashid [15]. For brevity, in the rest of this paper we will refer to this work as AA. For AA three non-trivial aspectual systems were studied, to discover the assumptions made by the different modules about the functionality, presence and implementation of other modules. The AA paper starts a catalogue of such assumption types, based on the assumptions discovered in the three case studies. All of the developers' assumptions found in the code were implicit: there is no denotation or documentation that states the meaning of the assumptions, and there are no rules laid out that codify how to ensure that the assumption is met. Hence, to discover the assumptions, the investigation consisted of manual inspection of the source code and developer interviews.

The AA text also proposes a followup that, to the best of our knowledge, has not previously been performed. It consists in codifying the assumptions

such that these can be "used to semi-automatically identify assumptions in other aspect code" [15]. This would allow, on the one hand for implicit assumptions to be elicited from the source code, and on the other hand for known 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" [15]. In this section we show how GASR can be used to perform exactly this. We implement elicitation rules for a subset of these implicit aspect assumptions and run them on two of the three case studies used in AA[5]. 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 [15]) and run the experiments on the HealthWatcher [37] and MobileMedia [38] systems. We implemented analysis rules for all implicit assumptions that can be sufficiently formalized into a set of rules, or approximated by a heuristic, codified as a set of rules. All these rules were developed on a test-first basis and both the rules and the test cases are available online[6]. 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[7]. 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, even on assumptions that are implicit in the source code.

We do not fully document all the analyses we created for assumption identification, for brevity. Instead we choose to focus here on the interesting analyses: those that achieve fully automatic verification, reveal new assumption instances and show customizability.

5.1. Assumptions on concretisation of pointcuts

The first assumption we talk about here was already mentioned in Sect. 2.3: an abstract pointcut that is concretized in a subclass and re-concretized in one of its subclasses. The implicit assumption is that in such a case the aspect actually would wish 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)))
```

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

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

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

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, which is in accordance to the results published in AA.

5.2. 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 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 add declarations of precedence between different aspects. One precedence assumption stated in AA is that the implicit precedence rules of the language are not modified by the precedence declarations that are explicitly stated by the developers. 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 by the developer, 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.

Note that development of the rules included thorough testing using a comprehensive test suite, where violations are indeed detected when present. Not finding any violations in both these case studies is simply a consequence of using these specific case studies. We consider that a search for a case study that does violate these rules would consist in an large amount of work that would however not add any significant value to the research. Hence we have not undertaken such a search.

5.3. 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 general assumption relies on the implicit meaning of "correct" functioning and therefore cannot be unambiguously made explicit, *i.e.,* defined in code rules that capture all the requirements of presence of other aspects. 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 the use of marker interfaces: the interface sometimes is stand-alone, *i.e.,* defined in its own compilation unit. Moreover, aspects may refer to a subinterface 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 identify the known instances of this assumption. But 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 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. The fact that these assumption instances have been overlooked in AA clearly demonstrates the benefits of automated analysis as enabled by GASR.

5.4. 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 relies on an implicit meaning: "mutually exclusive". Hence it also cannot be unambiguously defined in a code rule. At most we can infer some heuristics that can 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: having the same pointcut name and matching on the same join point shadows.

We do not claim that this conjecture and the resulting heuristics are particularly efficient, nor even valid. These only serve as an illustration of how we can use GASR to (attempt to) make implicit assumptions explicit and establish whether the software under study adheres to these assumptions. An arguably better solution would be to make such assumptions explicit in the source code, *e.g.,* through annotations, and base verification of the assumptions on those annotations. This will be accomplished in Section 6.

5.4.1. Same Pointcut Name

For this heuristic we assume that the name of the pointcuts convey their semantics. 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 because many aspects are used to implement a command pattern and they 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 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 because 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 it allowed for filtering out a high number of false negatives. This resulted in the discovery of six cases of copy-paste reuse, which is arguably not a good characteristic of the code.

5.4.2. 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 was not mentioned in the AA paper due to space limitations, but is present in the raw data.

5.5. 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" [15], so the implicit 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. 4.2. 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 not present in the raw data of AA. Hence this is again a new discovery made thanks to GASR.

5.6. Conclusion

In this section we have illustrated the usefulness of GASR by implementing detection of aspect assumptions (AA) that are implicit in the code. We have implemented logic rules for the detection of inter-aspect assumptions that have been elicited in earlier work [15]. This effectively achieves some of the future work laid out in that publication. We have run these rules on two of the three systems that were used for discovering the aspect assumptions. The results of these GASR queries were verified for correctness and completeness by manually inspecting the source code as well as by cross-checking with the full published list of assumption instances.

All the assumption instances that had been previously found were detected using GASR. More important though are the three following results: First, we achieved fully automatic verification of assumption instances in Section 5.1 and 5.2. Second, we also detected three previously unknown assumption instances in Section 5.3 and one in Section 5.5. Third, we have shown in Section 5.4.1 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.

6. Detecting Explicit Aspect Assumptions with GASR

Section 5 showed one use of GASR for analysing aspect-oriented programs. However, the automated analysis introduced there may only tell us that a particular assumption *might be made* by a given piece of aspect code, or that code

may violate an assumption, for example as discussed in Section 5.4. This analysis cannot confirm that such an assumption *is indeed being made* by the author of that code or that the code *indeed violates* the assumption. The only way in which we can ensure we capture the assumptions of the developers and to verify these assumptions is to provide developers with ways of expressing their assumptions directly in the code and have detection rules that rely on this explicit information. As we have already noted in our previous work on aspectual assumptions [15](AA), we could only be sure of assumptions identified from manual code analysis after discussing these initial findings with the original developers.

In the work on AA we discovered various implicit assumptions that required the use of some form of annotations for them to be checkable. Using such annotations together with GASR code analysis would increase both the maintainability and the reusability of the aspect code:

- *For maintainability:* any changes in the aspect's context (*e.g.,* the base code) can be automatically re-checked against the explicit assumption annotations. This way, any changes that will break aspect assumptions can be quickly identified and the aspect code can be adjusted accordingly.

- *For reusability:* When aspect code is written in a generic manner allowing its reuse in the context of different applications (*e.g.,* as is the case with Glassbox [39], one of the three projects studied in [15]), it becomes especially important that its assumptions can be validated for each specific reuse context.

In this section, we present a set of annotations that allows developers of aspect code to make explicit some of their implicit assumptions. These annotations are underpinned with GASR queries, enabling us to validate that the assumptions hold in any context in which the aspect code might be deployed. As in Section 5, rules were developed on a test-first basis and both the rules and the test cases are available online[8]. Also as in Section 5, we analyzed two of the three case studies used in AA and after running the analyses the results were verified for correctness and completeness.

Below, we discuss two types of annotations: We first discuss annotations that enable expressing assumptions on the static structure of an AspectJ program, in particular the presence or absence of specific types. We then describe some annotations that enable expressing more dynamic assumptions on the application control flow. Our annotations do not cover all assumption types from [15]. Other annotations could be expressed, but we do not consider them here due to space limitations. For each of the annotations that we introduce, we present the actual annotation as well as GASR rules that interpret the annotations and identify violations of the assumptions expressed. When analyzing the code, GASR will provide a warning if the assumptions do not hold.

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

6.1. Annotations for the Presence of Aspects

We first present the three different annotations we defined and then we show how they are used to verify assumptions on actual case studies. Each of the three annotations can be placed on a type and take as argument a list of type patterns or a list of label names (both of these were introduced in Section 4.2.2). Each annotation signifies the following:

- @Requires: If the type is present in the deployment, the types referred to in the argument should also be present.
- @Excludes: If the type is present in the deployment, the argument types may not be present.
- @OneOf: Exactly one of the argument types is present in the deployment.

Below we discuss some examples from the HealthWatcher and MobileMedia case studies to explain the use of these annotations as well as their implementation in GASR. For simplicity, we will only use type patterns here. The use of labels is shown in Section 6.2 and a note on labels versus patterns is given in Section 6.2.3.

6.1.1. Inclusion assumptions of aspects

In the MobileMedia system, the PhotoAndMusicAspect has as an assumption that the MusicSelector and PhotoSelector aspects have also been deployed, but the VideoSelector aspect has not. Instead, the PhotoAndMusicAndVideo aspect has been created for when all three photo, music and video aspects are present. This assumption for PhotoAndMusicAspect can be made explicit by annotating the aspect declaration as follows:

```
1 @Requires(type = "*..MusicSelector","*..PhotoSelector" )
2 @Excludes(type = "*..VideoSelector")
3 public aspect PhotoAndMusicAspect...
```

This uses a combination of the @Requires and @Excludes annotations to explicitly denote the assumption. GASR can then check for violation of the assumptions using the two logic rules detailed below.

First, the rule missing|required-requires reveals the names of types that have been declared to be required but are missing, together with the type that contains the annotation:

```
1(defn missing|required-requires [?name ?requirer]
2  (fresh [?reqds ?required]
3           (requiring|type-key-val ?requirer "type" ?reqds)
4           (contains ?reqds ?name)
5           (fails (type-type|pattern ?required ?name))))
```

In this rule, line 3 is similar to the rule for the @Label annotations we described in Sect. 4.2.2. It matches on the @Required annotations, and ?reqds will bind to the array of arguments of the type attribute. (For matching on labels,

the label attribute is used.) Through backtracking line 4 will successively bind ?name to each element of the array. Note that ?name may be a type pattern. Line 5 then succeeds if the pattern matching rule type-type|pattern cannot find a type ?required for that pattern, *i.e.,* the required type is missing.

Second, the rule present|excluded-excluder reveals the types that are present yet should not be, together with the class that contains the annotation, as follows:

```
1(defn present|excluded-excluder [?exd ?excluder]
2  (fresh [?exds ?excluded]
3           (excluding|type-key-val ?excluder "type" ?exds)
4           (contains ?exds ?exd)
5           (type-type|pattern ?excluded ?exd )))
```

Structured like the first rule, the only elements of note here are lines 3 where the @Excludes annotations are matched, and line 5 that succeeds if the pattern can be matched in the system, *i.e.,* if an excluded class is present.

Surprisingly, running these rules on the MobileMedia system reveals that the inclusion assumption is being violated. This is because on a default build of the system, all classes and aspects are compiled and included in the deployment. As a result, the VideoSelector aspect is also present, yet it should not be. As GASR respects build configurations (see Section. 4), to satisfy the rule it suffices to have a build configuration that does not include VideoSelector. This shows that simply downloading and building the MobileMedia system as-is will cause erroneous behavior, which was not clear from the download instructions.

6.1.2. Mutual Exclusion Assumptions

The HealthWatcher system is a client-server environment where the clients use Java RMI to connect to the servers. The system has however been designed to also allow for other communication methods: the abstract aspect HWClient-Distribution defines one pointcut that sub-aspects should use to redirect calls over the network. The RMIClientDistribution aspect is currently the only sub-aspect of HWClientDistribution. If another subaspect is created, deployment of the system should only include one of the subclasses, otherwise the same calls will be redirected multiple times over the network, once for each subaspect.

To make explicit this assumption, the following annotation is added to the declaration of HWClientDistribution:

```
1 @OneOf(type = "*..HWClientDistribution=")
2 public aspect HWClientDistribution...
```

This states that there may be only one subclass of HWClientDistribution present in the system. To verify this, GASR uses the following rule, revealing for which type declaration a @OneOf annotation is violated and which are the offending types:

```
1(defn tooMany|definer-offenders [?dec ?offs]
2  (fresh  [?patterns ?count ?pattern]
3           (oneOfing|type-key-val ?dec "type" ?patterns)
4           (contains ?patterns ?pattern)
5           (typepatterns-matches ?pattern ?offs)
6           (differs ?count 1)
7           (equals ?count (count ?offs))))
```

In this rule, line 3 binds `?patterns` to the collection of type patterns in a @OneOf annotation. Line 5 then constructs a set `?offs` for each pattern `?pattern` in `?patterns`. For example, for the annotation above there is one pattern and its set would contain all of the subclasses of HWClientDistribution. Lines 6 and 7 state that the number of elements in this set must be different from 1, *i.e.,* there are zero or more than one of the stated types present in the system.

Since there is only one subaspect of HWClientDistribution in the HealthWatcher system, we find that this mutual exclusion assumption is satisfied. If in an evolution of HealthWatcher a different distribution technology would be added, this rule will show that the mutual exclusion assumption has been violated. To solve this, different build configurations will need to be created; one for each distribution technology.

6.1.3. Assumptions on the use of Inter-Type Declarations

Returning to the MobileMedia system, there is an inter-type declaration (ITD) assumption of interest as it illustrates that the oneOfViolation rule needs to be able to handle lists of type name patterns. The MusicSelector aspect provides an advice that implements some of the handling of the selection of music. As part of its implementation it retrieves music album data, which is set by calling an ITD that is defined in the same aspect. Put differently, there is an assumption of a call protocol between an ITD and an advice: the ITD has been called before the advice executes, otherwise the music album data would be null.

This ITD is called in the PhotoAndMusicAndVideo and PhotoAndMusicAspect aspects, as part of the startup of the system. Hence, if either of these two aspects is present the assumption is satisfied. This can be made explicit by the following annotation on the MusicSelector aspect declaration:

```
1 @OneOf(type = "*..PhotoAndMusicAndVideo","*..PhotoAndMusicAspect")
```

For this annotation, in the oneOfViolation rule, typepatterns-matches (on line 4) constructs a set that comprises the types in the system whose name matches at least one of the patterns in the list of type names. Concretely, it will contain the PhotoAndMusicAndVideo and PhotoAndMusicAspect aspects.

As in Section 6.1.1, running this rule in GASR shows a violation: both the PhotoAndMusicAndVideo and PhotoAndMusicAspect aspects are present in the system, which means they initialize the music album data twice. Both these aspects are actually mutually exclusive, a deployment of the system should only include one. This assumption can also be made explicit, using @Excludes annotations in both aspects: each one excluding the other one.

While this use of the @OneOf annotation allows for verifying the assumption, it is not resilient to certain changes. If in an evolution of the code a third aspect is added that initializes the music album data, it will need to be added to the annotation. Alternatively, if from the existing two aspects the initialization code is removed, the @OneOf annotation no longer captures the requirement of the call protocol. A better solution would be to verify the control flow between the ITD and the advice, ensuring that the ITD is always called before the advice runs. This can also be verified using GASR, as we will show next.

6.2. Control Flow Annotations

We defined two annotations for verifying control flow assumptions. We present them here and then show their use. Each of these annotations is placed on a piece of behavior (a method, constructor or advice) and takes as argument a list of label names. Only label names can be used here because we need to be able to refer to advice and these do not have unique signatures, like constructors or methods. When analyzing the code, GASR will provide a warning if the requirements that are denoted using the annotations are not met. Each annotation signifies the following:

- @ExcludesPrevious this behavior may only execute if none of the argument behaviors have executed previously.
- @RequiresPrevious this behavior may only execute if all the argument behaviors have executed previously.

Similar to the above, GASR implements these annotations with the following rules:

```
1 (defn present|excludedPrevious-excluder  [?excluded ?excluder ?label]
2   (all
3            (exclPrev|behavior-val ?excluder ?label)
4            (labeled|behavior-label|val ?excluded ?label)
5            (behavior-reachable|behavior ?excluded ?excluder)))

6 (defn missing|requiredPrevious-requirer-label [?required ?requirer ?label]
7    (all
8             (reqPrev|behavior-val ?requirer ?label)
9             (labeled|behavior-label|val ?required ?label)
10            (fails (ajsoot/behavior-reachable|behavior ?required ?requirer))))
```

The first rule obtains the @ExcludesPrevious annotations `?excluder` and the value of their label in line 3 and the @Label annotations `?excluded` and their values in line 4. Both values need to be the same, and line 5 states that there is a control flow that arrives in `?excluder` after having passed through `?excluded`. This is a violation of the assumption, and the rule exposes both behaviors and the value of the label. The second rule is much alike to the first rule, except in the last line where it states that there is a control flow that did not pass through the label, violating the assumption.

GASR is more powerful than what these annotations can express. For example, we could extend the use of labels to explicitly require/exclude:

- a specific method to be invoked (in at least one, or in all potential executions of the program);
- a specific sequence of methods to be invoked;
- the receivers of successive method invocations to be the same object (in at least one potential execution of the program) or different (in all potential executions of the program); or
- the potential run-time types involved in an invocation (arguments, receiver) to include / exclude a specific subtype of the statically declared ones.

However it's not clear that any of these richer capabilities are indeed needed in practice. Using the annotations and rules we have defined, we can already successfully codify all of the 12 cases raised in [15]. In fact, we have only needed to use @RequiresPrevious, as we shall see in the remainder of this section.

6.2.1. Assumptions on the use of Inter-Type Declarations

As we have discussed in Section 6.1.3, in the MusicSelector aspect of MobileMedia there is an assumption of a call protocol between an ITD and an advice. Music album data must be set before it can be read and therefore the ITD always needs to be called before the advice executes. This assumption can be made explicit by annotating the ITD with @Label("SetMusicAlbumData") and annotating the advice with @RequiresPrevious("SetMusicAlbumData").

Source code analysis with GASR reveals that currently the system does satisfy the assumption, as already was established in Section 6.1.3. This annotation is however much more robust than the OneOf annotation used in Section 6.1.3 as it will not be affected by the code-evolution scenarios discussed in Section 6.1.3.

6.2.2. Assumptions on Sequential execution of Advice

Control flow checks should not only work on advice and methods, they should also be able to verify the flow between different advice. For example, in MobileMedia there is such an assumption in the PhotoAspect aspect. This aspect adds the functionality for viewing photos. To do so it adds a "View" command to the user interface when the menus are initialized in a first advice, and in a second advice intercepts menu pick actions, adding the foto viewing behavior. Hence, for the second advice to be able to work, the first advice needs to have run beforehand.

This assumption is made explicit by annotating the first advice with @Label("ViewCommandAdded"), and the second advice with @RequiresPrevious("ViewCommandAdded"). Using GASR we established that this assumption indeed holds.

Note that, even though these two pieces of advice are located in the same aspect, it is not obvious that they will be invoked in the right order. In fact, their invocation order depends on where in the base (and, potentially, any other

aspects deployed) their respective pointcuts match and in which order the corresponding join points are encountered during program execution. Checking the assumption, thus, requires a whole-system analysis and cannot be done manually in a robust fashion.

6.2.3. A Note on Label Annotations

Control flow annotations required label annotations so that we could precisely refer to various points in the execution of an AspectJ program that we wanted to relate. It is worth emphasising that we can of course use label annotations with the same label string in a number of places in our program, essentially stating that these are similar in relation to a particular assumption we want to express. Using backtracking, GASR will iterate through all occurrences of the same label, effectively ensuring the constraint holds for all places thus annotated. This makes label annotations potentially more flexible than wildcard types: annotations with the same label can be placed on types with vastly different names. Expressing similar quantification with type wildcards invariably leads to lengthy expressions.

At the same time, however, label annotations are potentially less robust than type wildcards: Because the location of constraint definition and the location of the label annotation(s) is different (possibly in different files), it is easily possible for a label to be missing from a newly introduced file or for a newly introduced label to accidentally break a constraint definition.

6.3. Conclusion

In this section, we have illustrated the use of GASR for defining annotations that enable developers to explicitly express their aspect assumptions. We have shown a number such annotations, their implementation in GASR, and their application in the HealthWatcher and MobileMedia case studies. We have only illustrated a selection of annotations; others can be defined to cover more of the catalogue from [15]. In particular, we have only focused on aspect-aspect–coordination assumptions and have only touched on assumptions on control or data flow.

7. Discussion

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

Completeness. The fact that we only performed our experiments on two concrete cases comprises a threat to validity. 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 [15]. As highlighted in Section 3, 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 4.2, 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.

Input Language. GASR is a source code analysis tool that works, as-is, on ASPECTJ source code only. Yet almost all published analyses of aspectual source code have been performed on ASPECTJ code [2, 3, 4, 5, 6, 8, 9, 11, 12, 13], and it is commonly considered as being the most used aspect-oriented language. As a result, GASR is a suitable alternative for these approaches and can also be used to analyse a substantial fraction of all aspectual source code. Moreover, the predicate library is relatively language-agnostic as it works in terms of the aspect-oriented concepts and the data obtained from the annotations. 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 [40]. Hence, in our opinion, GASR truly is general-purpose.

Performance. The logic engine of GASR is hardly optimized for speed, and as a consequence GASR query results are not instantaneous. Nonetheless, we found the performance of GASR to be satisfactory overall. This is thanks to our implementations of key library predicates dispatching on whether they are invoked with particular arguments bound or not, and some predicates calling out to Clojure or Java where an imperative approach is more efficient. Future work could consider switching to a more performant logic engine altogether. For instance, to the highly optimized LogiQL [41] on top of which en entirely logic-based implementation of a sophisticated points-to analysis [42] has outperformed imperative ones.

As for queries written by program developers, procedural knowledge about backtracking and unification in logic programming can drastically impact query evaluation times. For instance, logic conditions should be ordered such that the ones pruning most candidate solutions are evaluated first.

Another determining factor is the kind of predicates used in a query. In general, solutions to queries about the structure of the code are returned near-instantly. This is because the structural predicates from Section 4.2.1 and Section 4.2.2 merely have to consult the state of the ASPECTJ compiler (e.g., backtrack over the elements of a collection containing all advice known to the compiler). The same goes for most behavioral predicates from Section 4.2.3 that consult static analysis results provided by the SOOT [32] framework. Graph traversal predicates, however, should be used with care. Insufficiently restricted invocations easily enumerate all paths between arbitrary nodes in the program's control flow graph. For applications where this is desirable, a dedicated model checker may perform better.

Usability and Existing Uses. To the best of our knowledge, there are no studies on how application developers use program queries in practice. There has been

empirical research on the kind of questions developers frequently ask about there code [43, 44, 45]. Using a program querying tool would be appropriate to answer some of these questions. For instance, a question that involves several variables would require extensive configuration of such a tool. Alternatively, when several separate tools would need to be combined to answer the question, using just one general program querying tool would be better.

To answer questions about AspectJ code, Gasr users need to compose and launch a program query from the appropriate predicates and inspect its results. The approach we have used in this text is is to follow a set of established (design or code) patterns or assumptions and translate these into a logic form. Examples of this are in the logic rules for the reentrancy example and the wormhole pattern (both in Section 4.2) as well as the different rules for the different aspectual assumptions (in Sections 5 and 6). While this approach is a good starting point for describing rules and queries, it is by no means the only way of working with Gasr and we expect users to use a variety of approaches, depending on the task at hand.

Constructing queries for Gasr requires familiarity with our query language, although Gasr's read-eval-print loop and graphical result inspector do facilitate this task (cf. Section 4). This concern is shared by all approaches to program querying that feature a dedicated query language. Their support for user-defined queries renders them more powerful than tools that are limited to answering pre-defined queries, yet the downside is that this power comes at the cost of requiring users to master the query language.

Several examples exist, however, of the use of such tools by experts as an enabling technology in development tooling and in empirical studies. The Rascal [46] language, for instance, powers the PHP AiR tool [47] which has been used to study the usage of PHP features in a large corpus [48]. The .QL language [49] is powering a commercial quality assessment tool suite that computes metrics and violations of coding rules [50]. Ekeko [31] powers academic prototypes for tracing [51] and automating [52, 53] changes to source code, and has been used in a large corpus study about the maintenance of automated functional tests [54]. Soul [29] powers, among others, a tool for enforcing design regularities in code [55] or for performing annotation-aware refactorings [56]. Each of these languages features dedicated support for expressing the characteristics of source code elements that need to be retrieved. This renders them attractive for prototyping development tools. Comparisons of the expressiveness of some of these languages exist [57, 58], but they are not yet comprehensive.

The most likely users of Gasr will therefore not be application developers, but rather tool builders. In fact, we are already aware that the aspectual frame inference tool ajFX[9] [59] builds on top of Gasr for computing which variables may potentially be modified during the execution of an advice. Along the same lines, the predicates from Section 5 for detecting potential implicit inter-aspect assumptions could be packaged as a tool on their own. The same goes for the

[9]Available from `https://github.com/timmolderez/ajfx`

predicates from Section 6 that verify assumptions made explicit in the code through annotations. This tool would be tasked with launching pre-defined queries, composed of the predicates we have presented, and presenting the user with their results. As such, its use would not require any particular expertise from an application developer —but be limited to checking the set of predefined assumptions presented in this paper.

8. 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 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 5.4.1.

We state that what is required is a general-purpose aspectual source code analysis tool that reasons about source code and its metadata, as present in annotations. Also, the source code reasoner should be extensible and customizable to the code at hand. This will allow to avoid duplicate work building analyses and furthermore enable existing analyses to be reused, extended and customized to fit 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: an ASPECTJ source code analysis tool in the tradition of logic querying. GASR is a General-purpose Aspectual Source code Reasoner for ASPECTJ-like languages whose analyses may be customized relatively straightforwardly, as illustrated in Section 5.4.1 and take annotations into account, as shown in Section 6. 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. First we detected a subset of implicit inter-aspect assumptions that were previously identified by Zschaler and Rashid [15] by manual inspection of the same software. The same work had revealed assumptions that were implicit, so second we made them explicit through the use of annotations and verified that they were satisfied. 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 in their work.

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 [15] which we did not implement yet are an avenue for further work. Thirdly, our control flow reasoning currently is limited to relating two points on an execution path. This is less powerful than the LTL approach of MAVEN [26], which naturally supports relating multiple points on the same path. Launching a composite query for related control flow annotations would bring us closer to the LTL approach, but is not yet implemented. Lastly, GASR can be seen as base infrastructure on which new and more advanced analyses can be built. Our preferences are for analyses that can extract design-level documents [60] and provide information on whether there exist any dependencies and interactions between aspects [61].

Acknowledgments

Johan Fabry is partially funded by FONDECYT project number 1130253, Coen De Roover is partially funded by the *Cha-Q* SBO project sponsored by the "Flemish agency for Innovation by Science and Technology" (IWT Vlaanderen). Carlos Noguera is funded by the FWO AIRCO project. Thanks to Romain Robbes for feedback on draft versions of this text. The original research on aspect assumptions was partially funded by the European Union under Marie-Curie fellowship RIVAR.

Bibliography

[1] J. Fabry, C. De Roover, and V. Jonckers, "Aspectual source code analysis with GASR," in *Proceedings of 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'13)*, 2013, pp. 53–62.

[2] 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.

[3] J. Wloka, R. Hirschfeld, and J. Hänsel, "Tool-supported refactoring of aspect-oriented programs," in *Proceedings of the 7th International Conference on Aspect-oriented Software Development(AOSD '08)*. New York, NY, USA: ACM, 2008, pp. 132–143.

[4] 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.

[5] R. Khatchadourian, A. Rashid, H. Masuhara, and T. Watanabe, "Detecting broken pointcuts using structural commonality and degree of interest," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*. Lincoln, Nebraska, USA: IEEE press, nov 2015.

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

[7] 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.

[8] 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.

[9] 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.

[10] J. Zhao, "Change impact analysis for aspect-oriented software evolution," in *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE '02)*. New York, NY, USA: ACM, 2002, pp. 108–112.

[11] 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)*, 2008, pp. 36–47.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] J. Fabry, A. Kellens, S. Denier, and S. Ducasse, "AspectMaps: Extending Moose to visualize AOP software," *Science of Computer Programming*, vol. 79, pp. 6 – 22, 2014, http://dx.doi.org/10.1016/j.scico.2012.02.007.

[21] E. Poll, P. Chalin, D. Cok, J. Kiniry, and G. T. Leavens, "Beyond assertions: Advanced specification and verification with JML and ESC/Java2," in *In Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, volume 4111 of LNCS*. Springer, 2006, pp. 342–363.

[22] M. Papi, M. Ali, T. Luis Correa, J. Perkins, and M. Ernst, "Practical pluggable types for Java," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2008, pp. 201–212.

[23] S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble, "JavaCOP: Declarative pluggable types for Java," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 2, pp. 1–37, 2010.

[24] A. Kellens, C. Noguera, K. De Schutter, C. De Roover, and T. D'Hondt, "Co-evolving annotations and source code through smart annotations," in *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society Press, 2010, pp. 119–128.

[25] M. Goldman, E. Katz, and S. Katz, "MAVEN: Modular aspect verification and interference analysis," *Formal Methods in System Design*, vol. 37, no. 1, pp. 61–92, 2010.

[26] E. Katz and S. Katz, "User queries for specification refinement treating shared aspect join points," in *Proc. 8th IEEE Int'l Conf. Software Engineering and Formal Methods (SEFM'10)*, J. L. Fiadeiro and S. Gnesi, Eds., 2010, pp. 73–82.

[27] 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.

[28] 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.

[29] 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.

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

[31] C. De Roover and R. Stevens, "Building development tools interactively using the Ekeko meta-programming library," in *Proceedings of the IEEE CSMR-WCRE 2014 Software Evolution Week, Tool Demo Track (CSMR-WCRE14)*, 2014.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] D. Pickering and R. Bodkin, "Glassbox project page," Published on-line: http://sourceforge.net/projects/glassbox/, accessed April 09, 2015.

[40] 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.

[41] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[42] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA09*, vol. 44, no. 10, pp. 243–262, 2009.

[43] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE10)*, 2010, pp. 175–184.

[44] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of Software Engineering (FSE06)*, 2006.

[45] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Evaluation and Usability of Programming Languages and Tools (PLATEAU10)*, 2010, pp. 8:1–8:6.

[46] P. Klint, T. v. d. Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM09)*, 2009.

[47] M. Hills and P. Klint, "PHP air: Analyzing PHP systems with rascal," in *Proceedings of the IEEE CSMR-WCRE 2014 Software Evolution Week, Tool Demo Track (CSMR-WCRE14)*, 2014, pp. 454–457.

[48] M. Hills, P. Klint, and J. J. Vinju, "An empirical study of PHP feature usage: a static analysis perspective," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA13)*, 2013.

[49] O. de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble, "Keynote address: .ql for source code analysis," in *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM07)*, 2007, pp. 3–16.

[50] P. Avgustinov, A. I. Baars, A. S. Henriksen, R. G. Lavender, G. Menzel, O. de Moor, M. Schäfer, and J. Tibble, "Tracking static analysis violations over time to capture developer characteristics," in *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE15)*, 2015.

[51] C. N. Angela Lozano and V. Jonckers, "Managing traceability links with matraca," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER16), Tool Demonstration Track*, 2016.

[52] C. De Roover and K. Inoue, "The Ekeko/X program transformation tool," in *Proceedings of 14th IEEE International Working Conference on Source Code Analysis and Manipulation, Tool Demo Track (SCAM14)*, 2014.

[53] T. Molderez and C. De Roover, "Automated generalization and refinement of code templates with ekeko/x," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER16), Tool Demonstration Track*, 2016.

[54] L. Christophe, R. Stevens, C. De Roover, and W. De Meuter, "Prevalence and maintenance of automated functional tests for web applications," in *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSMe14)*, 2014.

[55] J. Brichau, A. Kellens, S. Castro, and T. D'Hondt, "Enforcing structural regularities in software using IntensiVE," *Science of Computer Programming*, vol. 75, no. 4, pp. 232–246, 2010.

[56] C. Noguera, A. Kellens, C. De Roover, and V. Jonckers, "Refactoring in the presence of annotations," in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM12)*, 2012.

[57] R.-G. Urma and A. Mycroft, "Programming language evolution via source code query languages," in *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools*, 2012, pp. 35–38.

[58] Z. Ujhelyi, Á. Horváth, D. Varró, N. I. Csiszár, G. Szőke, L. Vidács, and R. Ferenc, "Anti-pattern detection with model queries: A comparison of approaches," in *Proceedings of the IEEE CSMR-WCRE 2014 Software Evolution Week (CSMR-WCRE14)*, 2014.

[59] T. Molderez, "Modular reasoning in aspect-oriented languages," PhD, Universiteit Antwerpen, Antwerp, 10/2014 2014.

[60] 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.

[61] 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.