Code Querying by UML

Carlos Noguera, Coen De Roover, Andy Kellens, Viviane Jonckers

Software Languages Lab,

Vrije Universiteit Brussel, Belgium

Email: {cnoguera,cderoove,akellens,vejoncke}@vub.ac.be

Abstract—The need to identify source code that exhibits particular characteristics is essential to program comprehension. In this paper we introduce ARABICA, a tool for querying Java code using UML class and sequence diagrams. Our use of UML diagrams avoids the need for developers to familiarize themselves with yet another language. In contrast to tools that rely on dedicated query languages, ARABICA encodes querying semantics in a dedicated, minimal UML profile. Stereotyped class and sequence diagrams, characterizing structural and behavioral properties respectively, are translated into logic program queries. Using examples from the JHotDraw framework, we illustrate the utility of ARABICA in validating design invariants, finding design pattern implementations and exploring extension points. We present a pre/post-test quasi experiment as a preliminary assessment of our approach.

I. Introduction

Code querying is an important activity in program understanding. The ability to answer questions about the code base of a system is useful whenever developers are confronted with an unknown system, prior to making changes to an existing system, or when determining whether idioms and coding conventions have been adhered to. However, most Integrated Developments Environments (e.g., Eclipse) provide developers only limited support for asking questions about their code and having them answered. The code querying functionality of mainstream IDEs is restricted to answering predefined queries such as finding the method declaration that corresponds to a method invocation, or computing the hierarchy in which a class resides. Support for formulating and answering custom, complex or composite queries is limited or plain lacking.

More advanced code querying functionality is provided by certain extensions to IDEs or third-party tools that allow the developer to formulate custom queries. Nevertheless, query tools that model the code as a stream of characters (i.e., grep-like tools) suffer from a lack of expressive power. Others require developers to learn a query language with foreign syntax and semantics (e.g., ASTLog [3], LePUS [9]). These languages seldom allow developers to retrieve complex structures (e.g., design pattern implementations) with straightforward queries.

To address these concerns regarding custom code querying tools, we introduce ARABICA as a graphical tool —embedded in the Eclipse IDE— that uses UML diagrams as queries for querying Java code. The goal of ARABICA is to provide an expressive query tool that allows developers to describe both structural and behavioral characteristics of the code they search for using a familiar language. Moreover, ARABICA's graphical nature aims at easing the description of complex structures.

To this end, ARABICA extends the UML meta-model with a minimal profile composed of seven stereotypes that allow developers to use class and sequence diagrams as templates to search for code. The stereotypes mark UML model elements as variables in a query, or extend the semantics of relations to express transitivity or absence.

Using model-to-text tools, class as well as sequence diagrams are translated into a SOUL [6] query that is executed to find the solutions to the original UML query. ARABICA gains its expressive power from this SOUL back-end, which includes the ability to reason over the structure and behavior of a program by consulting the static analyses computed by SOOT [23].

The remainder of this paper is structured as follows. First, we motivate the need for specialized querying tools like ARABICA through three examples taken from the JHotDraw code base. Section III discusses ARABICA by first explaining the syntax and semantics of the SOUL program query language, then introducing ARABICA's extensions to the UML meta-model and finally discussing how UML entities can be interpreted as queries by illustrating their transformation into SOUL queries. Having explained the functioning of our approach, we revisit the three motivating examples in Section IV, where we show the UML class and sequence diagrams that serve as queries for their solution. Section V presents a preliminary study into the usability of ARABICA, by means of a pretest-posttest quasi-experiment with 11 participants which reveals a positive attitude towards our tool in general, and class diagrams as a means to describe the structural qualities of a query in particular. Finally, Section VI presents related work, and we conclude in Section VII.

II. Motivating Example

We illustrate the need for a code querying tool like ARABICA using three examples from JHotDraw[1]. JHotDraw is a two-dimensional graphics framework for structured drawing editors that is written in Java. The framework is meant to showcase the utility of design patterns in constructing extensible frameworks. As with any framework, developers must gain a thorough understanding of its structure and design invariants before they can make use of it. We consider three examples of queries that developers could formulate over the code base of JHotDraw: how figures are implemented,

---

[1]http://www.jhotdraw.org

whether tools are correctly implemented with regards to design invariants, or what instances of a particular design pattern (e.g., a Composite) can be found in the framework.

A. Implementation of Concrete Figures

Figures are key to the JHotDraw framework. When constructing a graphical editor, developers need to implement classes that represent the editor's specific figures. Figures in JHotDraw are implemented as subclasses of the `AbstractFigure` class. As a novice developer is confronted with this task, looking at how existing figures are implemented can provide insight. Nevertheless, finding suitable examples might be difficult, for example, JHotDraw figures can be further specialized into *connection figures*. Thus, in order to find examples, the developer must *find all the concrete subclasses of `AbstractFigure` that do not implement the interface `ConnectionFigure`.*

B. Correct Activation of Tools

Figures in JHotDraw are constructed and manipulated through *Tools*. Tools act as a modifier of the mouse behaviour on the drawing: all mouse events are forwarded to the active tool. As such, tools can be *activated* or *deactivated* through methods defined on the `Tool` interface. Developers can define their own tools by extending the `AbstractTool` class and overriding its methods to implement the additional functionality. When doing so, developers must take care to respect the design invariants of the framework. One such design invariant, that is described in the documentation of `AbstractTool`, is that subclasses overriding the `AbstractTool#activate()` method must, in their body, delegate to `super.activate()`. JHotDraw developers might therefore wish to verify the correctness of their implementation by formulating a code query that identifies violations of this design invariant. This query would take the form: *which classes extending `AbstractTool` and overriding the `activate()` method do not call `super.activate`?*
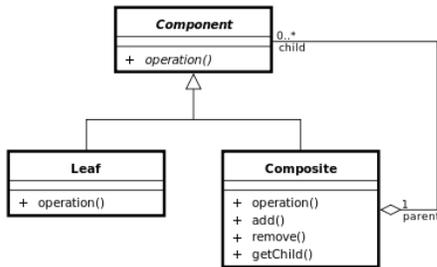
C. Composite Pattern

Fig. 1.   Structural characteristics of the Composite pattern

As JHotDraw employs design patterns heavily, developers using the framework must acquaint themselves with their implementation. In particular, complex figures use the Composite design pattern, delegating operations such as the calculation of

the bounding box (i.e., the minimum rectangle that encloses the figure) to each of the components of the figure. This calculation is defined by the `displayBox()` method of the `AbstractFigure` class. The developer can find this, and other uses of the Composite pattern by formulating a query that describes the structural (Figure 1) and behavioral (e.g., the delegation of `Composite.operation()` to `Component.operation()` characteristics of the pattern.

Although IDEs such as Eclipse provide primitive support for code querying, this support is insufficient to express queries such as the ones required in the three examples above. Without a specialized querying tool, the developer, for example seeking concrete figures, must ask the IDE to show all subclasses of `AbstractFigure` and then search manually for concrete ones, and then filter out those that implement `ConnectedFigure`.

III. ARABICA

ARABICA is a tool for querying Java code that uses UML models as queries. In ARABICA, UML class and sequence diagrams specify respectively the structural (i.e., classes, fields, methods and their relations) and the behavioral characteristics (i.e., messages sent between methods) of the sought after code.

For UML models to be used as code queries, it is necessary to extend the UML meta-model. To this end, ARABICA provides a UML profile that defines a number of stereotypes. Figure 3 depicts the `ArabicaProfile` that contains seven concrete stereotypes extending relationships, named elements, operations, properties and comments with the information required to use them as queries.

As a tool, ARABICA extends the Barista framework [6] for querying Java code using the SOUL program query language. We reuse Barista's Query Result View to enable developers to obtain solutions to a query either one at the time or all at once. From this view, it is also possible to inspect the solutions to a query and navigate to the corresponding points in the code. This integration with Barista also allows us to schedule ARABICA queries to be run in the background every time a particular Java project is built. We embedded the editors provided by TOPCASED[2] in Barista's query editor as depicted in Figure 2. TOPCASED is a fully featured UML modeling environment, based on Eclipse's Modeling Framework, that provides graphical editors for class and sequence diagrams. However, any UML editor that produces models conforming to the EMF implementation of the UML meta-model can be used. ARABICA translates such a UML model into a corresponding SOUL query that is evaluated against the program under investigation.

ARABICA, BARISTA and the SOUL tool suite are freely available and can be downloaded from: http://soft.vub.ac.be/SOUL/download-soul/

In the following sections, we recapitulate the syntax and semantics of the SOUL program query language before in-
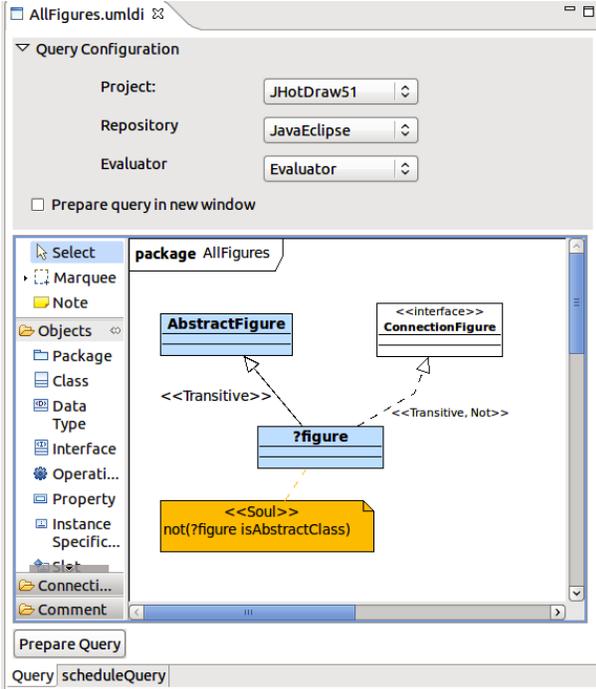
---

[2]http://www.topcased.org/

Fig. 2. ARABICA's query editor

troducing ARABICA's `ArabicaProfile` and its translation from UML diagrams to a SOUL query.

A. SOUL

SOUL [6] is a logic-based language for querying Eclipse Java projects. A SOUL query consists of conditions that quantify over the project's AST nodes. These conditions express the characteristics of the sought after code. We will briefly describe the syntax and semantics of the kind of conditions that ARABICA relies on. The following query consists of three conditions:

```
if ?class isClassDeclaration,
   ?class extends: ?super,
   ?super classDeclarationHasName: {.*Exception}
```

SOUL queries start with the keyword `if`. Logic variables are preceded by a question mark. SOUL's syntax for predicates closely resembles the one of Smalltalk, similar to a message that is sent to the first argument of the predicate. For each subclass of `ASTNode` (e.g., `ClassDeclaration`), SOUL provides a unary predicate that quantifies over the AST nodes of that kind (`isClassDeclaration:/1`). Thus, the variable `?class` in the first condition ranges over all class declaration nodes. Binary predicates quantify over the relations between two AST nodes. For instance, binary predicate `extends:/2` quantifies over all classes and the super class they extend directly. The second condition therefore binds `?super` to the direct super class of `?class`. In the third condition, the regular expression `{.*Exception}` substitutes for the name of this super class. Under SOUL's unification procedure, this condition succeeds for names that match the regular expression. SOUL features several extensions to Prolog's unification

procedure that are specific to the domain of code querying [5], [6]. Apart from these extensions, SOUL finds solutions to a query in the same manner as Prolog (i.e., through SLD-resolution [10]). The solutions to the above query therefore consist of all classes `?class` such that the name of their super class `?super` ends in `Exception`.

ARABICA also relies on conditions that use SOUL's template terms and negation predicate **not**/n. They are illustrated by the following query. Its solutions comprise a class `?class` that is instantiated by an instance creation expression `?exp`, even though all of its field member declarations are static:

```
if ?class classDeclarationHasName: ?className,
   jtExpression(?exp){new ?className(?argList)},
   not(jtClassDeclaration(?class){
        class ?className {
            ?member := [?modList ?type ?fieldName;]
        }
   },
   not(?member isStatic))
```

The query consists of three conditions. The first condition uses binary predicate `classDeclarationHasName:/2` to bind `?className` to the name of a class `?class`. The second condition uses a template term that consists of a functor (i.e., `jtExpression`) followed by an argument (i.e., `?exp`) and a code excerpt that is demarcated by braces. The functor of the template term identifies the grammar rule adhered to by the code excerpt. This grammar describes the concrete syntax of Java, extended with logic variables and a minimum of non-Java syntax. Used as a condition, a template term succeeds if the term's argument unifies with an AST node that matches the code excerpt. Variables within the excerpt are unified with the corresponding children of the match. As a result, the second condition quantifies over all expressions `?exp` that instantiate a class named `?className` using arguments `?argList`.

SOUL's negation predicate **not**/n implements Prolog's negation as failure. The predicate succeeds if the query composed of its arguments does not have any solutions. The third condition above therefore succeeds if `?class` does not have a field declaration `?member` that is not static. The template term illustrates some of the non-Java syntax that can be used within code excerpts. Operator `:=` unifies the logic variable on its left-hand side (i.e., `?member`) with the AST node that matches the code within square brackets on its right-hand side (i.e., a field declaration).

Note that SOUL does not match AST nodes with template terms in a strict, syntactic manner. Indeed, classes with nothing but a single field declaration are rare. Instead, SOUL lends template terms an example-driven semantics [5]. A matching AST node has to exhibit all characteristics exemplified by the template (i.e., the field declaration), but is allowed to exhibit additional ones (e.g., a method declaration). Moreover, the matching process recognizes implicit (i.e., implied by the semantics of Java) implementation variants of the exemplified characteristics. To this end, SOUL's aforementioned domain-specific unification extensions consult whole-program analyses computed by the SOOT [23] Java optimization framework.

For instance, a semantic analysis allows the bindings for each occurrence of *?className* to deviate syntactically (e.g., a fully qualified and an unqualified name in the second and third condition respectively), as long as they denote the same class declaration *?class*. The bindings for variable occurrences that substitute for expressions are, similarly, allowed to differ syntactically (e.g., `this.field` and `this.getField()`) as long as they may evaluate to the same object at run-time according to a points-to analysis. The latter is relied upon by the SOUL queries that ARABICA generates for UML sequence diagrams (cf. Section III-C).

B. UML Extensions

The Arabica UML profile extends the semantics of UML models such that they can be used as specifications of a code query. To this end, `ArabicaProfile` defines a minimal set of stereotypes (depicted in Figure 3), divided in three groups: SOUL-constraints, Relationships and Variables.

1) SOUL-Constraints: First of all, `ArabicaProfile` extends the `Comment` meta-class of the UML meta-model with a «Soul» stereotype. This enables including comments in a UML diagram that represent a regular SOUL condition. Comments with the stereotype «Template» can be included in a UML class diagram to specify the structure of a property or an operation through a code template. To this end, the meta-classes for `Operation` and `Property` are extended with a stereotype «Templated» that has a property `template` linking the operation or property to its corresponding code template. In a similar manner, «Soul» comments can be attached to relations, variables of templated operations, as specified in the profile by the «SOULConstraint» abstract stereotype.

2) Variables: Users of ARABICA have be able to specify which parts of a UML diagram represent logic variables. To this end, `ArabicaProfile` extends the `NamedElement` UML meta-class. `NamedElements` represent all elements that carry a name (e.g., classes, interfaces, operations and messages). The «Variable» stereotype is applied to elements defined as sub-types of `NamedElement` such that they might act as variables in a query.

To facilitate specifying queries, ARABICA includes the convention that all elements in a class or sequence diagram whose name starts with the character '?' are to be treated as if they carried the «Variable» stereotype, this follows the SOUL manner of specifying logic variables.

3) Relationships: Finally, `ArabicaProfile` provides stereotypes «Not» and «Transitive» to adapt the semantics of relations. Both stereotypes extend the `Relationship` meta-class. This meta-class represents all structural relations in UML models such as inheritance and association relations between classes. Stereotyping a relation with «Not» specifies that the relation should not be present in the queried code, whereas stereotyping a relation with «Transitive» specifies that the related elements must be transitively reachable.
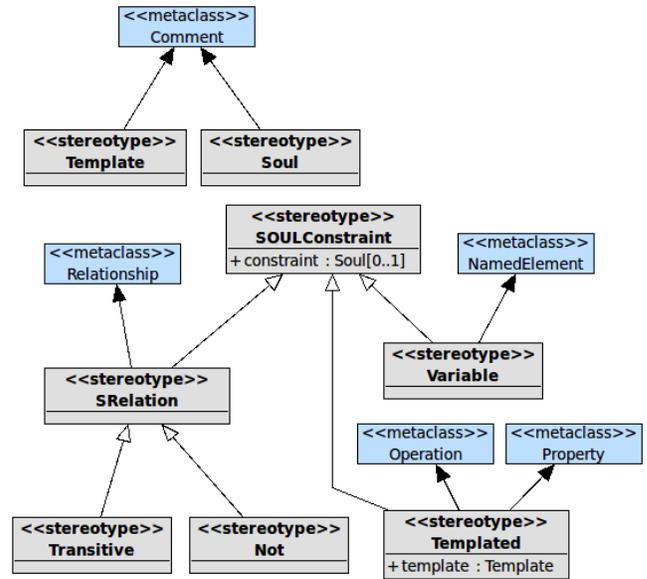

Fig. 3.   Arabica UML Profile

C. Translation to SOUL

Through a translation into SOUL queries, we specify ARABICA's query semantics for UML models on which the `ArabicaProfile` is applied. ARABICA produces a SOUL query from such a model using the ACCELEO[3] model-to-text transformation engine. Table I depicts the SOUL queries that correspond to selected entities from UML class and sequence diagrams.

ARABICA translates a UML model by recursively visiting its packages. First, the individual classes from the class diagram are translated together with their relationships. Next, the collaborations defined in the sequence diagrams are translated.

1) Class Diagram Queries: The first row of Table I illustrates the translation scheme for classes. Each class in a class diagram is translated into a `jtClassDeclaration` template term that binds Java class declaration AST nodes to the logic variable *?ADecl*. The name of the variable is obtained by appending `Decl` to the name of the class. A similar naming convention is used to capture the declaration nodes from fields and methods within the class.

Inheritance relations between classes and interfaces are translated into a condition that requires either the `isSubTypeOf:/2` or the `extends:/2` predicate to hold for the corresponding type declaration variables, depending on whether the inheritance relation is transitive or direct. This is illustrated by the second row in Table I. Had the «Not» been applied to either of the inheritance relations in row two, its effect would have been to surround the resulting SOUL conditions with the **not/n** predicate.

The only association type currently supported in ARABICA is directed simple associations such as the ones depicted in the third row of Table I. Such an association is translated

differently depending on the cardinality of its second side. If the second side has a cardinality of one (e.g., the one labeled *?b* in Table I), then the association is translated as a field in the template for the owning class.[4] If the cardinality of the association is unbounded (e.g., the one labeled *?bs* in Table I), the generated field is required to be a subtype of `java.util.Collection`. Note that the generated conditions do not ensure that this collection will actually carry objects of type B. However, future incarnations of ARABICA could incorporate other implementations of such associations (e.g., array-based ones).

Rows four and five of Table I illustrate that UML comments containing SOUL code are inserted directly in the generated query.

2) Sequence Diagram Queries: Sequence diagrams are translated in a two-step process. First, each lifeline is translated into a `jtExpression` template term that binds the name of the lifeline (*?b* in the last row of Table I) to its corresponding instance creation expression. The idea is to capture the expression that gives birth to the lifeline and subsequently unify this expression with the receiver of each method invocation directed towards the lifeline —thus differentiating between different lifelines of the same type. SOUL's domain-specific unification procedure (cf. Section III-A) unifies two syntactically differing expressions (i.e., an instance creation expression and the receiver of a method invocation) if they may evaluate to the same object at run-time according to a points-to analysis computed by SOOT [23]. If ARABICA detects that SOOT has not been run for the queried project, no such `jtExpressions` are generated.

Next, a `jtMethodDeclaration` template term is generated for each message targeted at a lifeline. The name of this method corresponds to the name of the incoming message. For every outgoing message in the activation block, a method invocation is added to the body of this template term. SOUL's example-driven matching of template terms (cf. Section III-A) allows method declaration AST node *?operationDecl* to match multiple of such `jtMethodDeclaration` templates that each contain one or more of its statements.

Note that variables *?operationDecl* and *?method* already occurred in the first and fourth row of Table I. They effectively link the sequence and class diagrams of the UML model together. As a result, the depicted sequence diagram specifies an interaction between the classes A and *?class* of the class diagram.

IV. MOTIVATING EXAMPLE REVISITED

Having explained the implementation of ARABICA, we now turn back to the three use cases sketched in Section II, showing for each one its implementation as a UML query. To give an idea of the running time for each query, Table II lists the number of milliseconds taken by ARABICA to provide all results, the number of results, and the time to reach the

---

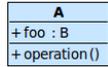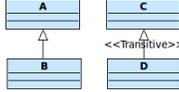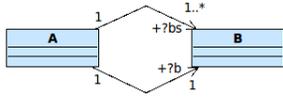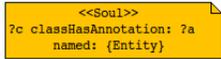[4]For readability purposes we have omitted the capturing of the field declarations.

UML Entity	SOUL conditions

A
 + foo : B
 + operation()
	`jtClassDeclaration(`*?ADecl*`){`
	 `    public class A{`
	 ?fooDecl `:= [B foo;]`
	 ?operationDecl `:=`
	 `    [void operation(){`
	 `        } ]`
	 `    }`
	 `}`

A, C, B, D with <<Transitive>>	*?BDecl* `extends:` *?ADecl*`,` *?DDecl* `isSubtypeOf:` *?CDecl*

A — B, 1 / 1..* +?bs, 1 / 1 +?b	`jtClassDeclaration(`*?ADecl*`){`
	 `    class A{`
	 `        B` *?b*`;`
	 `        `*?collection* *?bs*`;`
	 `    }`
	 `},`*?collection* `isSubTypeOf:`
	 `{java.util.Collection}`

?class
 <<Templated>>+?method()
 <<Template>> throw new RuntimeException();
	`jtClassDeclaration(`
	 `    `*?classDecl*`){`
	 `    class `*?class*`{`
	
	 ?methodDecl `:=[`
	 `    public void `*?method*`(){`
	 `    throw new`
	 `        RuntimeException();`
	 `    }]`
	 `    }`
	 `}`

<<Soul>> ?c classHasAnnotation: ?a named: {Entity}	*?c* `    classHasAnnotation:` *?a* `    named: {Entity}`

?a : A, ?b : ?class
 operation() → ?method()
	`jtExpression(`*?b*`){`
	 `    new `*?class*`(`*?argList*`},`
	
	 `jtMethodDeclaration(`
	 `    `*?operationDecl*`){`
	 `    public void operation(){`
	 `        `*?b*`.`*?method*`();`
	 `    }`
	 `}`

TABLE I
SUMMARY OF THE TRANSLATIONS FROM SELECTED UML ENTITIES TO SOUL

first solution when run against JHotDraw version 5.1. It is important to note that these times are not a full evaluation of the performance of our approach, which remains the subject of future work.

Query	One Result	All Results	# of results
Example 1	232	3702	21
Example 2	90	199	2
Example 3	41353	516752	24

TABLE II
RUNNING TIME FOR THE QUERIES, ALL TIMINGS ARE GIVEN IN MS.

A. Implementation of concrete figures

Figure 4 illustrates how the query "*Find all the concrete subclasses of* `AbstractFigure` *which do not implement the interface* `ConnectionFigure`." can be ex-
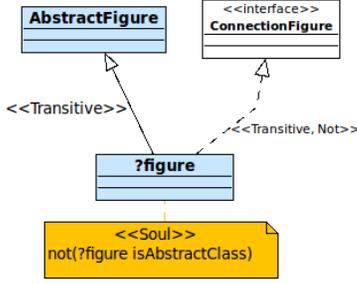
Fig. 4. Query to find implementations of Figures which are not Connection-Figures

pressed in ARABICA. The stereotypes «Transitive» and «Not» are used to find a *?figure* which is a sub-class of AbstractFigure and not of ConnectionFigure. A SOUL comment has been added to the class diagram to restrict *?figure* to concrete classes. This is needed because both an abstract and a concrete class can match the corresponding jtClassDeclaration template term under SOUL's example-driven matching (i.e., what is not specified in a template term, cannot constrain its matches).
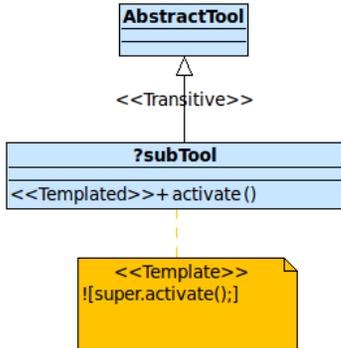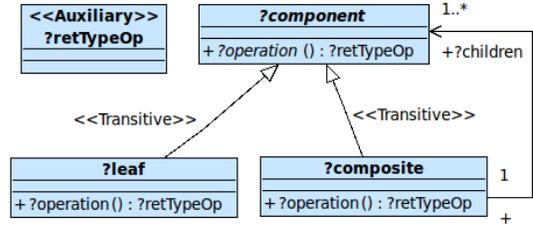
B. Correct Activation of Tools

Fig. 5. Query to find violations to the design invariant: activate of sub-tools must delegate to super
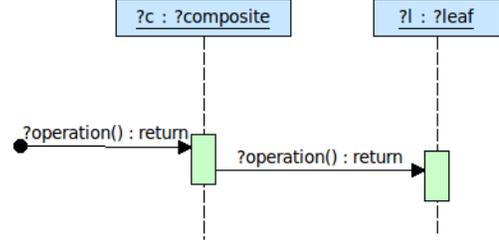
The UML class diagram depicted in Figure 5 corresponds to the query "*which classes extending AbstractTool and overriding the activate() method do not call super.activate?*". It can be used to have ARABICA find violations of the design invariant that governs the correct activation of tools in JHotDraw. The UML query specifies the implementation of faulty activate() methods by means of a template associated with the operation in the *?subTool*.

C. Composite Pattern

Figure 6 depicts an ARABICA UML query that looks for implementations of the composite pattern. The query consists of two parts: a class diagram specifying the structure of the

(a) Structural query of Composite Pattern

(b) Behavioral query of Composite Pattern

Fig. 6. Arabica query to find uses of the Composite Pattern in JHotDraw

pattern, and a sequence diagram denoting that classes implementing the role of a *composite* must delegate an operation to *leaf* classes.

The class diagram (Figure 6(a)) closely resembles the prototypical implementation presented in Figure 1. Classes *?leaf* and *?composite* indirectly extend a *?component* class, and override its abstract *?operation*. The query associates *?composite* classes with *?component* classes by means of a *?children* variable. An auxiliary class named *?retTypeOp* has been added to the diagram to serve as the return type for the *?operation*. Auxiliary classes, marked as such by a «Auxiliary» stereotype taken from the standard profile offered by TOPCASED, do not appear in the solutions to the UML query. To this end, ARABICA does not translate them into jtClassDeclaration terms.

The sequence diagram (Figure 6(b)) specifies the behavioral properties of the pattern. The diagram states that whenever an instance *?c* of *?composite* receives a message *?operation*, it must delegate this message to an instance *?l* of the class matched as *?leaf* (within its implementation of the method).

With this query, we were able to find that the displayBox():Rectangle method in the hierarchy of AbstractFigure does in fact implement a Composite pattern (in which StandardDrawing takes the role of the *composite* class).

V. PRELIMINARY STUDY OF USABILITY OF ARABICA

To assess the feasibility of querying source code using UML class and sequence diagrams, we performed a pre-experimental user study with 11 participants. As this kind of study is a quasi-experiment, as opposed to a full scientific experiment, it does not allow us to make any founded claims regarding the usability of ARABICA. However, it does provide

insights into how potential users of ARABICA perceive and value the various features of the query tool. In literature, such studies have been successfully applied for providing an initial assessment of program comprehension tools [17].

A. Study design

The study is conceived as a pretest-posttest quasi-experiment. This experiment consists of a pretest that quantifies the expectations of a participant regarding program query tools before being introduced to ARABICA. After having used the tool, participants are required to fill out a posttest that measures their perception of the tool. By comparing the results of the pretest and the posttest, we can quantify how exposure to ARABICA influenced the participants' perception of UML-based queries, and which of ARABICA's features were deemed useful by the participants. To this end, we measured the following properties:

- **Value of graphical query languages:** Do graphical query languages provide any added value?
- **Ease of understanding:** Are the UML-based queries easy to understand?
- **Class diagrams:** Do class diagrams provide a suitable means to express structural characteristics?
- **Sequence diagrams:** Do sequence diagrams provide a suitable means to express behavioral characteristics?

Each participant of the study was invited for a session that took between 30 and 45 minutes in total. At the start of this session, the participants were asked to fill out the pretest. This pretest consisted of 20 statements that, next to measuring the properties mentioned above, enquired about the participant's background knowledge. The participants scored each of the statements on a 5-point Likert scale.

Afterwards, the participants received a short demonstration of the features of Arabica. Following this demonstration, the participants were asked to use ARABICA to identify all incorrect activations of tools in JHotDraw (the example discussed in Section II-B). To do this, they were handed a step-by-step tutorial that guided them throughout the process of formulating the query and verifying its results. Once they were finished with this task, they were asked to fill out a posttest consisting of 19 statements. These statements measured the perception of the participants of the task executed during the session, their evaluation of the use of UML class and sequence diagrams as a means to query source code, and their valuation of the different features of ARABICA.

Due to space limitations, we are not able to include the pretest and posttest in this paper. Both questionnaires, all the material offered to the participants, the script followed by the test conductor, as well as the 11 filled out pre/posttests is available on-line[5].

B. Participants profile

The 11 participants of our user study are all experienced computer researchers with various backgrounds. 7 of the
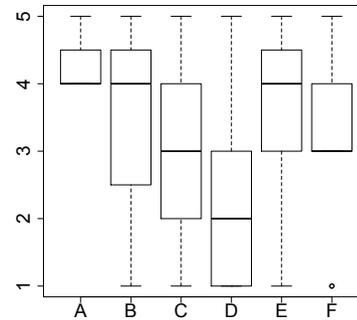
[5]http://soft.vub.ac.be/ICPC2012/

Fig. 7. Boxplot of the background of the participants. (A) Development experience (B) Java experience (C) Eclipse experience (D) Knowledge UML (E) Understand class diagrams (F) Understand sequence diagrams.

participants hold a Masters degree and the remaining 4 a PhD. During the pretest we enquired them about their knowledge of development in general, Java, the use of Eclipse, and UML. Figure 7 provides a summary of this enquiry. As can be seen in boxplot (A), all of the participants rate themselves as experienced developers. With the exception of one participant, all of the participants have some experience with Java (B) and Eclipse (C); with a median of 4, most participants have indicated to be proficient with both Java and Eclipse. As for their knowledge of UML, all but one of the participants consider themselves to only have limited knowledge of UML. This is however contrasted with their perception of the understandability of UML class and sequence diagrams. All participants find both kinds of diagrams relatively easy to understand.

C. Observations

1) Pretest-posttest: Figure 8 provides an overview of results of the pretest and posttest. Overall, ARABICA was well-received by the participants. As can be seen in Figure 8(a), most participants were positive with respect to the use of a graphical query language. Only two of the participants did not find the use of a graphical query language to have any added value. However, we can observe that the perception of the value of graphical query languages has improved considerably in the posttest. A similar observation can be made for the ease of understanding (Figure 8(b)). In the pretest, the participants expressed their reservations regarding the understandability of graphical queries. Despite this negative appreciation of such queries expressed in the pretest, the posttest revealed that the participants found ARABICA's queries to be easy to understand.

As for the use of UML class and sequence diagrams as a means to query source code, the difference between the pretest and posttest is not as large. Most participants agreed that class diagrams are a suitable means to express structural characteristics. Using sequence diagrams to express behavioral characteristics was not well-received: the majority of the participants did not find them useful. If we compare

(a) Value of graphical query language

(b) Ease of understanding

(c) Class diagrams
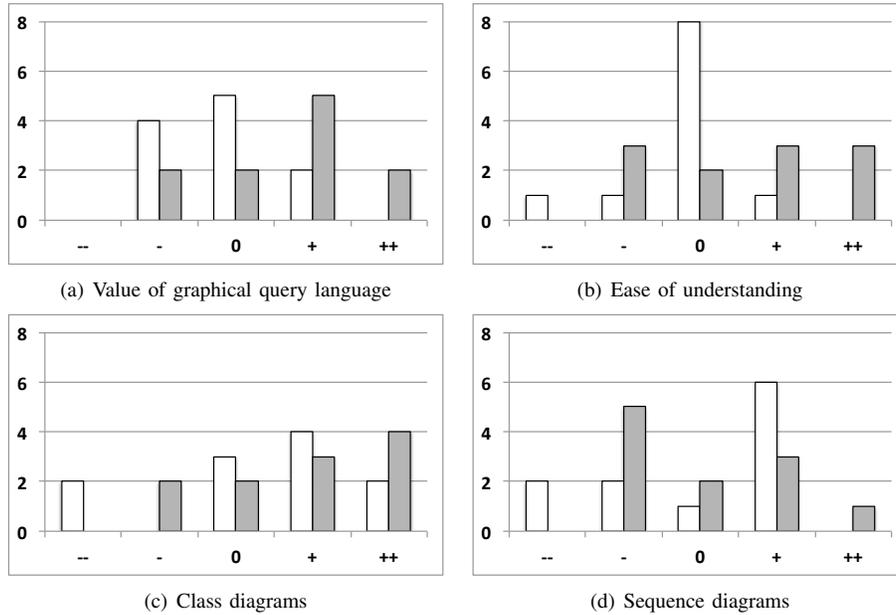

(d) Sequence diagrams

Fig. 8. Comparison of the pretest (indicated in white) and posttest (indicated in grey). X axis depicts the 5-point Likert scale, Y axis is the number of participants that selected each point.

the average appreciation of using sequence diagrams between the pretest and the posttest, we see that the exposure to our tool did not have an impact. This was also confirmed in our discussions with the participants: they expressed the fear that using sequence diagrams would not scale beyond simple examples. Consequently, we consider it future work to identify other kinds of diagrams that allow expressing behavioral characteristics in an intuitive and scalable fashion.
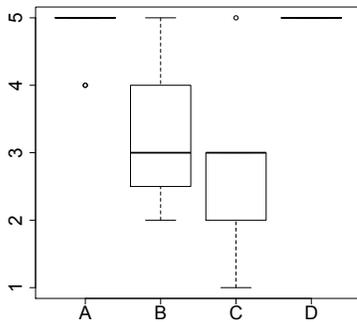

Fig. 9. Appreciation of the features of ARABICA. (A) Use of the «Transitive» stereotype (B) Query result view (C) One by one access to results (D) Navigation link to code.

2) Features of ARABICA*:* The posttest also included statements regarding the usefulness of some of the features of ARABICA. Figure 9 presents, by means of boxplots, a summary of the participants' evaluation of these features. UML diagrams do not provide a means to express that relationships between entities can be transitive (e.g., extending directly or indirectly from a class). To provide support for char-

acterizing such relationships ARABICA makes use of the «Transitive» stereotype. While we were not certain that users would find this approach intuitive, Figure 9(A) illustrates that all participants of the study valued it. Similarly, the participants unanimously rated the fact that the source code corresponding to the results of a query can be navigated directly from ARABICA as a very useful feature (boxplot (D)).

Regarding the effectiveness of the representation of the results of queries, the opinions of the participants were divided (boxplot (B)). Currently, ARABICA presents a query's results in either a tabled or a tree view. Interviews with the participants revealed that this representation could be made more intuitive by adopting a representation similar to that of Eclipse's search functionality that shows the results of a query grouped according to the package and class structure of the system under investigation. Finally, ARABICA provides functionality to compute the solutions to a query one-by-one. This feature was added such that for queries that are computationally intensive, it is possible to retrieve individual results instead of having to wait until all solutions were computed. As can be seen in boxplot (C), the participants of the study did not find this feature useful. One of the comments received from the participants was that they would prefer to be able to execute the query in the background and get results as they are computed.

D. Threats to validity

As mentioned above, our pre-experimental study does not allow us to make any generalizable claims regarding the usability of ARABICA. Nevertheless, the user study does allow us to observe how potential users perceived our tool. The validity of these observations is however subject to a number

of threats to validity.

One risk associated with our study is that the tutorial executed by our participants is overly simplistic or does not align to a realistic usage scenario of ARABICA. To assess this risk, our posttest included a number of statements that measured the participants' view on the performed task. While the participants did not assess the task as being too difficult, in general they did not find it overly simplistic. More than half of the participants agreed with the statement that the task was similar to queries they need answered during their day-to-day development activities. Given that we wanted to keep the amount of time necessary to execute the task manageable, the risk exists that the task does not capture the complexity associated with real-life queries.

Another risk lies within the composition of our group of participants: as all participants are computer science researchers they might not form a representative sample of software developers in general. Furthermore, they might be inclined to favor experimental tools, thereby rating ARABICA higher than they actually value it. First, despite that all participants are researchers, their background knowledge varies significantly: while all participants rate themselves as expert developers, they have different levels of experience with Eclipse, Java and UML. Therefore, we believe our user group not to be strictly homogeneous. Second, to mitigate the risk of an overly positive evaluation of ARABICA, we stressed the fact that we were expecting the participants to answer honestly, as this would otherwise void their contribution to the user study.

VI. RELATED WORK

a) Textual program query languages: A wide range of program querying languages can be found in literature. These languages offer a textual specification language to express structural or behavioral characteristics of a program, and identify locations in the code that exhibit these characteristics. For example, languages based on graph rewrite rules [18], logic formulas [1]–[3], [7], [14], [20], [26], and constraint programming [13] have been proposed to express structural characteristics. In order to express behavioral characteristics, we find approaches based on reachability queries [8], [24], [25], temporal logic formulas [15], state machines [11], and logic formulas [12], [16] that reason over the results of data flow and control flow analyses. Related to these approaches is also our previous work on SOUL [6]. This query language – that lies at the basis of ARABICA– offers a logic-based and example-based specification language for expressing structural and behavioral characteristics.

ARABICA is complementary to the above, *textual* approaches: by using UML class and sequence diagrams, it offers developers a *graphical* means for expressing program queries. Furthermore, ARABICA offers the full capabilities of such dedicated program query languages since it allows for embedding SOUL queries and code templates within the visual queries.

b) Visual program query languages: Within the database community, the use of visual languages to query databases has been well-established. Examples of these approaches are Query-By-Example [27], DOODLE [4] and PSQL [21]. While visual languages are not as common-spread in the domain of program querying, there exist a number of approaches that are closely related to ARABICA. In what follows we give an overview of these approaches.

LePUS [9] is a visual, formal specification language based on first-order predicate logic. Diagrams made using LePUS (called *Codecharts*) can be used to model and visualize object-oriented software systems. By means of tool support, it is possible to extract such Codecharts from existing Java source code, as well as to verify the validity of a Codechart with respect to the source code. While it is not one of the main goals of the approach, this last feature of LePUS allows it to be used to query source code.

Reiss presents *MURAL* [19], a simple visual language (equivalent to relational algebra) for querying over multiple data sources. A MURAL query consists of a number of entities (represented as boxes) and relationships between these entities (arrows between the boxes). Each entity comprises a set of fields that represent the domain that is being queried; relationships are expressed in terms of these fields. MURAL offers special constructs for restricting the entities and relationships of a query, along with support for composing queries and calculating transitive closures. While not limited to the domain of program querying, MURAL can be used to query code.

The main difference between these approaches and ARABICA lies in the visual notation that is being used: while both LePUS and MURAL offers their own, minimal visual vocabulary, our approach employs the well-known UML class and sequence diagrams as a visual notation for specifying queries. While all of these visual notations offer limited expressivity, ARABICA circumvents this limitation by allowing regular SOUL queries to be integrated with the visual notation. Furthermore, the fact that ARABICA queries are valid UML models results in that our approach is agnostic of the editors and tools that are used to create and manipulate queries.

Within the context of model-driven software engineering, Stein et al. [22] have proposed *Join Point Designation Diagrams (JPDDs)* as an alternative to OCL for querying UML models. JPDDs is a visual language that, similar to ARABICA, uses the extension facilities of UML for augmenting UML with a number of operators that enable its use as a query language. While ARABICA's notation as a consequence shares a number of similarities with JPDDs, both approaches differ fundamentally as JPDDs are aimed for querying UML models, while ARABICA queries the source code of programs.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we have introduced ARABICA, a tool for querying Java code using UML class and sequence diagrams. ARABICA provides a profile that extends entities from UML class and sequence diagrams such that they can be used to respectively specify structural and behavioral characteristics of sought-after code. Stereotyped diagrams are translated into logic queries expressed in the SOUL program query language.

The main advantage of our tool is that it offers developers a familiar, graphical language that allows for the expression of complex queries that yet remain easy to understand. By relying on UML we spare developers from having to learn a new language – in contrast to other program query approaches. To illustrate the use of ARABICA, we presented three examples taken from the implementation of the JHotDraw framework: implementation of figures, correct activation of tools and the implementation of the Composite design pattern. A preliminary evaluation of our tool was done by means of a pretest-posttest quasi-experiment with 11 participants. This evaluation revealed an overall positive attitude towards our tool. Participants found the use of UML class diagrams a suitable means to express structural characteristics. However, the participants were less receptive to the use of sequence diagrams to describe behavioral characteristics.

In future work, we plan to evaluate alternative UML diagrams for specifying such behavioral characteristics, starting by the investigation of activity and interaction diagrams. Furthermore, expanding on the positive evaluation of class diagrams as a means to query structure, we will explore means to express architectural characteristics using higher-level diagrams (such as packages and components). At a technical level, our tool suffers from a number of limitations, which was also confirmed by the user study. Amongst them, we plan to improve the representation of query results by integrating them with the Eclipse search engine and provide support to execute queries in the background. As mentioned in Section III-C1, ARABICA currently expects associations between classes to be implemented using `Collections`. We will diversify the implementations that ARABICA recognizes in future work. Finally, we will follow up on the preliminary evaluation of ARABICA by performing a controlled experiment with students.

REFERENCES

[1] M. Appeltauer and G. Kniesel. Towards concrete syntax patterns for logic-based transformation rules. In *Proceedings of the Eighth International Workshop on Rule-Based Programming (RULE07)*, 2007.

[2] T. Cohen, J. Y. Gil, and I. Maman. JTL: the Java Tools Language. In *Proceedings of the 21st annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA06)*, 2006.

[3] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the 1997 USENIX Conference on Domain-Specific Languages (DSL'97)*, 1997.

[4] I. Cruz. DOODLE: a visual language for object-oriented databases. In *International Conference on Management of Data*, pages 71–80, 1992.

[5] C. De Roover. A logic meta-programming foundation for example-driven pattern detection in object-oriented programs. In *Proceedings of the 27th IEEE International Conference on Software Maintenance, Post-doctoral Symposium (ICSM11)*, 2011.

[6] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers. The soul tool suite for querying programs in symbiosis with eclipse. In *International Conference on the Principles and Practices of Programming in Java (PPPJ)*, pages 71–80, 2011.

[7] K. De Volder. JQuery: A generic code browser with a declarative configuration language. In *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages (PADL06)*, 2006.

[8] S. Drape, O. de Moor, and G. Sittampalam. Transforming the .NET intermediate language using path logic programming. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'02)*, 2002.

[9] A. Eden. *Codecharts: Roadmaps and Blueprints for Object-Oriented Programs.* Wiley, 2011.

[10] M. H. V. Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)*, 23(4), October 1976.

[11] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI00)*, 2000.

[12] H. Falconer, P. H. J. Kelly, D. M. Ingram, M. R. Mellor, T. Field, and O. Beckmann. A declarative framework for analysis and optimization. In *Proceedings of the 16th International Conference on Compiler Construction (CC07)*, 2007.

[13] Y.-G. Guéhéneuc. *Un cadre pour la tracabilite des motifs de conception.* PhD thesis, Ecole des Mines de Nantes, June 2003.

[14] 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)*, volume 4067 of *Lecture Notes in Computer Science*, 2006.

[15] D. Lacey. *Program Transformation using Temporal Logic Specifications.* PhD thesis, University of Oxford, August 2003.

[16] 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.

[17] N. Matthijssen, A. Zaidman, M.-A. Storey, I. Bull, and A. van Deursen. Connecting traces: Understanding client-server interactions in ajax applications. In *International Conference on Program Comprehension*, pages 216–225, 2010.

[18] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE02)*, 2002.

[19] S. Reiss. A visual query language for software visualization. In *Symposium on Human Centric Computing Languages and Environments*, pages 80–82, 2002.

[20] T. Rho, G. Kniesel, and M. Appeltauer. Fine-grained generic aspects. In *Proceedings of the AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL06)*, 2006.

[21] N. Roussopoulos and D. Leifker. An introduction to PSQL: a pictorial structured query language. In *IEEE Workshop on Visual Languages*, number 77-87, 1984.

[22] D. Stein, S. Hanenberg, and R. Unland. A graphical notation to specify model queries for mda transformations on uml models. In *Model-Driven Architecture: Foundations and Applications*, pages 60–74, 2004.

[23] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON99)*, 1999.

[24] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *Proceedings of the 28th International Conference on Software Engineering (ICSE06)*, 2006.

[25] N. Volanschi. Condate: a proto-language at the confluence between checking and compiling. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP06)*, 2006.

[26] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, Belgium, January 2001.

[27] M. Zloof. Query by example: a data base language. *IBM Systems Journal*, 16(4):324–343, 1977.