Program Querying with a SOUL: the BARISTA tool suite

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*—Extracting information from the source code of a program is an important step in the way to program understanding, manipulation, development and maintenance. To this end, logic-based query languages provide a declarative manner in which to identify program elements of interest. In this paper we present BARISTA, a tool-suite for querying Java programs based on the Smalltalk Open Unification Language (SOUL). BARISTA offers programmers an advanced IDE to write queries and navigate their results. Tool builders can benefit from SOUL querying facilities by exploiting the on demand code querying and query scheduling services offered by BARISTA.**

I. INTRODUCTION

Program queries identify program elements that exhibit characteristics of interest. Logic formulas can be used as expressive and descriptive specifications of these characteristics. This merely requires reifying the program under investigation such that logic variables can range over its elements. Executing a proof procedure will then establish whether program elements exhibit the characteristics specified in a formula. This logic-based approach to program querying is widespread in literature (usually based on Prolog [6] or Datalog [3]). We will focus on querying Java programs using the latest incarnation of SOUL [8], one of the earliest logic program query languages that is still under active development.

Tool builders that require gathering information on the structure and behaviour of a system, can benefit from advanced program query facilities. Consider implementing a tool for a Java IDE that highlights repeated string concatenations where a `StringBuilder` would be more appropriate. Usually, this entails telling the IDE to mark some of its AST nodes. We therefore have to find those AST nodes that correspond to repeated string concatenations. Invoking the search API of the IDE is, on one hand cumbersome since the characteristics of the sough pattern must be described imperatively and on the other, only a feasible option in case the sought after AST nodes are characterized behaviorally. However, existing search APIs do not support control flow and data flow characteristics required to effectively track the values being concatenated across the execution of the application. Employing a logic-based query language would remedy the first shortcoming of IDE's search APIs, but provide no help regarding the second one. Furthermore, since most logic-based program query approaches represent the program's source code as a set of terms, it is up to the tool developer to translate the results of the query back to AST nodes for further processing. In its current incarnation SOUL addresses these problems by reasoning directly over the AST nodes of the application and allowing for the unification of terms based on static – control and data flow – analyses.

In this paper we present BARISTA, an Eclipse IDE plugin that allows tool developers to leverage the code querying facilities of SOUL to reason over Java code contained in Eclipse projects. The paper is divided in two parts: a short introduction to SOUL, its syntax and semantics, and an overview of its most prominent features. The second part explains the BARISTA tool-suite and the features it offers both to programmers and tool builders.

II. LOGIC PROGRAM QUERYING WITH SOUL

SOUL is a logic-based program query language in which users express the characteristics of the program elements that SOUL has to identify (e.g., all abstract methods) through logic conditions that quantify over a reified program representation. In this section, we briefly introduce the core syntax and semantics of SOUL in an informal manner. We restrict our discourse to the features that have changed since the earliest publications on SOUL, those that facilitate exploiting the solutions to a program query in software engineering tools and those that facilitate specifying a program query.

A. Syntax and Semantics

Like Prolog, SOUL finds solutions to a query using SLD-resolution. The syntax of SOUL differs slightly from the one of Prolog though. This is illustrated by the SOUL queries depicted in Figure 1. SOUL queries start with the keyword *`if`*. Logic variables are preceded by a question mark. In each solution to the query on line 1–2, variable *`?method`* is bound to an abstract-method declaration with a name bound to *`?name`*. The query consists of two conditions. The first condition uses

```
1  if ?method methodDeclarationHasName: ?name,
2     ?method isAbstractMethod

3  if ?m isMethodDeclaration,
4      [?m getParent] equals: ?t,
5      ?t typeDeclarationHasBodyDeclarations: ?l,
6      ?i equals: [?l lastIndexOf_Object: ?m]
```

Fig. 1. Logic program queries illustrating some unique features of SOUL.

the binary predicate `methodDeclarationHasName:/2`[1] to bind `?method` and `?name` to a method declaration AST node and its name. The second condition of the query uses the unary predicate `isAbstractMethod:/1` to restrict the bindings for `?method` to those methods marked as being abstract. Each pair of bindings for `?method` and `?name` constitute a single solution to the query. Both the predicates used in the query stem from the Cava library for reasoning about Java programs, included with the Soul distribution.

B. Linguistic Symbiosis and AST representation

Note that the program representation used by software engineering tools (static analyses, IDE) and logic program query languages do not necessarily align. The abstract grammars of their ASTs might differ or the tool might have canonicalized the source code before constructing its ASTs. Worse, the tool might not be using ASTs at all in its representation. This is true for most tools that are specialized in control flow and data flow characteristics and that use data structures such as control flow graphs. Finding the AST node that corresponds to a node in a control flow graph is hard. Entire sub-graphs may correspond to a single AST node (e.g., a `for`-statement). The nodes of the control flow graph might even stem from an intermediate representation (e.g., a three-address based one).

Soul overcomes this issue in two ways: First, the predicates of the Cava library only reify AST nodes. While some predicates consult program analyses that are computed for an intermediate representation, its elements will not show up in the solutions to a query. This way, tool builders are shielded from its intricate details and are not tasked with mapping intermediate code to source code themselves. Secondly,Soul's symbiosis with the underlying Smalltalk runtime [7] enables the Cava library to forego reifying AST nodes as compound terms. Instead, *the reified version of an Eclipse AST node is the AST node itself.* This is illustrated by the condition on line 4 of Figure 1 where the predicate `equals:/2` is used to unify variable `?t` with a Smalltalk term[2]. *Smalltalk terms* are delimited by square brackets and can contain logic variables wherever Smalltalk variables are allowed.

C. Domain-Specific Unification and Template queries

Example-driven matching of code templates [4], [5] is a unique feature of Soul that facilitates specifying program queries. It enables *exemplifying* source code characteristics through familiar *template terms* rather than specifying them through convoluted logic conditions. A template term consists of a functor (e.g., `jtClassDeclaration`) followed by an argument 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-native syntax. For instance, the template

[1]Predicates names are post-fixed with the number of parameters they take.

[2]A *Smalltalk term* (e.g., `[?m getParent]`) unifies with another term (e.g., `?t`) if and only if its expression evaluates to a value that unifies with the term.

```
1  if jtMethodDeclaration(?method){
2    abstract ?type ?name(?paramList);
3  }

4  if jtMethodDeclaration(?m){
5    public static void main(String[] args) {
6     ?scanner := new java.util.Scanner(?argList);
7     ?scanner.close();
8     ?scanner.next();
9    }
10 }
```

Fig. 2. Template queries

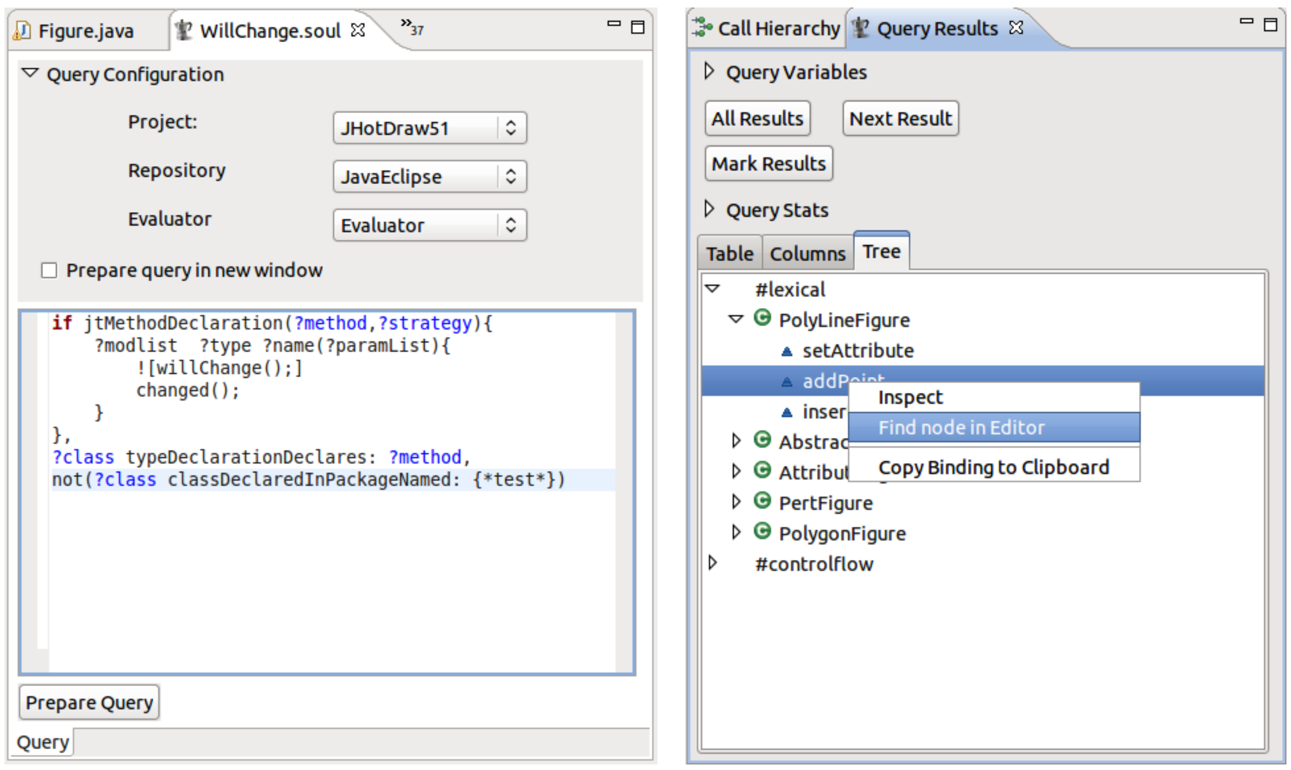
term on line 1–3 of Figure 2 exemplifies an abstract method, providing bindings for its return type, name and argument list in the logic variables `?type` `?name` and `?paramList` respectively.

Several matching strategies are employed when resolving a query. In the most strict strategy, AST nodes in the application are required to match the AST nodes in the template term. In the most lenient strategy, AST nodes in the application must exhibit the *control flow* specified in the template for it to match. The template term on lines 4–10 specifies a control flow in which values from a `?scanner` are read after the `?scanner` is closed. Under the strict strategy only methods that *exactly* match the template will be identified; however, under the most lenient strategy, the template will match if an execution path that starts from a `main` method in the application includes the three statements expressed in the body of the template. Notice that the three statements are not required to reside in the same method, but rather on different methods called transitively from the `main`. In addition to the control flow specified by the order of the statements in the template, repeated uses of the `?scanner` logic variable specify a *data-flow* dependency. In order to support these characteristics, Soul relies on domain-specific unification in which two Java expressions will unify if they are on a *may-alias* relation i.e., the two expressions may contain the same value at runtime.

III. Barista Tool-suite for Program Querying

Barista is a set of Eclipse plugins that serve as communication bridge between the IDE and Soul. Its goal is to serve as an IDE for querying Java programs from within the Eclipse IDE and to offer services to other plugins that require source code query facilities.

The Barista tool-suite is composed of three eclipse plugins: First, the Barista-core plugin offers a Java interface to the underlying Soul engine. Queries over the state of the source code under study are delegated through this interface to the Smalltalk instance that runs Soul. Symbiosis between Java objects representing the AST of the system and the Smalltalk-based reasoning engine is achieved by the JavaConnect library. Second, the Barista-UI plugin offers developers an IDE to write Soul queries and inspect its results. Finally, the Arabica plugin builds on top of the Barista-UI editor to provide a graphic syntax for Soul



Project: JHotDraw51
Repository JavaEclipse
Evaluator Evaluator

```
if jtMethodDeclaration(?method,?strategy){
    ?modlist  ?type ?name(?paramList){
        ![willChange();]
        changed();
    }
},
?class typeDeclarationDeclares: ?method,
not(?class classDeclaredInPackageNamed: {*test*})
```

```
#lexical
  PolyLineFigure
      setAttribute
      addPoint
      inser
  Abstrac
  Attribut
  PertFigure
  PolygonFigure
#controlflow
```

(a) Query Editor provided by the BARISTA-UI plugin

(b) Corresponding query result view

Fig. 3. Query editor provided by the Barista-UI plugin.

based on UML-class diagrams. Each of the components of the BARISTA tool suite are explained below.

A. BARISTA-*core*

The main plugin in the BARISTA tool suite is BARISTA-core. It is charged with interfacing between Eclipse plugins and the SOUL engine. To this end, two services are provided: on demand code querying and scheduling queries to be run whenever a particular Java project is built. These two services are realized by means of a Java interface and a plugin extension point respectively. In both cases, queries are fed to the BARISTA-core as strings, and their results are provided back as maps that go from logic variables present in the queries to their corresponding bindings for each solution found. It is important to note that the objects contained in the map are actual Eclipse JDT AST nodes, which makes the manipulation of the results (e.g., marking the results of a query on the Java editor) convenient.

B. *A* SOUL *IDE for Eclipse*

Writing SOUL queries can be a daunting task for developers. Thus, we provide an Eclipse plugin (BARISTA-UI) that serves as an IDE for writing SOUL queries and inspecting their results. The BARISTA-UI plugin extends the BARISTA plugin through the mechanisms explained in the previous section to provide a query editor and a query results view (Figure 3). Through BARISTA-UI, developers can execute queries on projects within their own workspace, serving as a general-purpose code querying facility, complementing that of the Eclipse JDT.

The BARISTA-UI plugin provides an advanced query editor complete with syntax-highlighting, auto-completion, syntax-error checking and contextual help. Figure 3(a) shows a query that finds violations to a protocol regarding change notifications, excluding methods in test packages. Violations to the protocol are specified by means of a code template, while the exclusion of test methods is specified using predicates that state that methods found cannot be contained in a sub-package of `test`. In addition to editing SOUL queries, it also serves as a launcher for queries. For this, the developer must specify (at the top of the figure) the project in which to perform it, the repository that contains the rules implementing the predicates and the evaluator to use. Once these choices are made, the query can be executed.

By clicking on the `Prepare Query` button (Figure 3(a) on the bottom), a *Query Results* view is launched (Figure 3(b)). This view allows the developer to control the execution of the query, asking the underlying SOUL engine for solutions either all at once (*All Results* button) or one at a time (*Next Result*). Additionally, the view provides statistics on the execution of the query, such as number of results found and time taken executing the query. Once the query is executed, bindings for each variable in each solution found are presented in three different visualizations: Table, Column and Tree mode. The developer can interact with each result found by means of

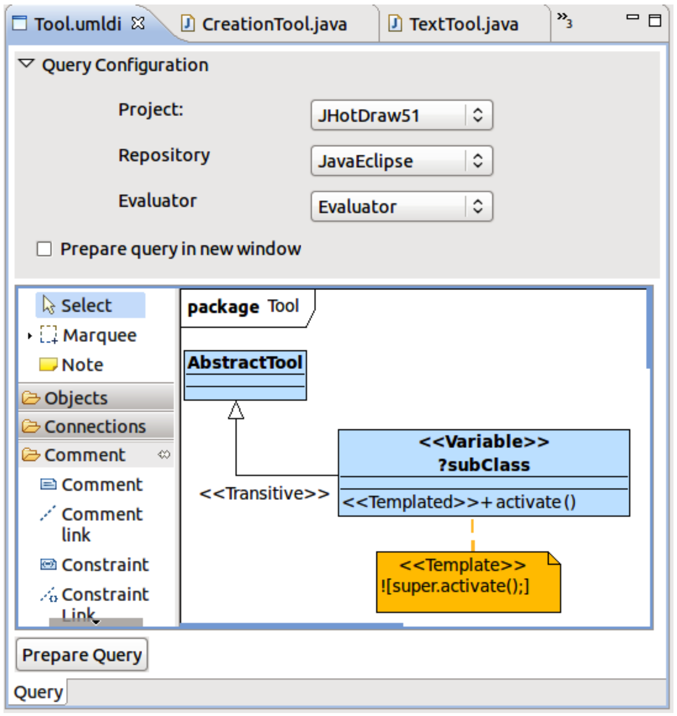


Fig. 4. UML query using ARABICA

a context menu. The menu offers to open an inspector on the node, open an Eclipse editor that contains the AST node represented in the solution, or copy a text representation of the binding to the clipboard. Finally, the developer can mark all resulting AST nodes using the Marker facilities provided by Eclipse.

C. ARABICA *UML-based querying*

The final plugin of the BARISTA tool suite is ARABICA. This plugin builds on top of the BARISTA-UI to offer an alternative syntax for specifying structural patterns in SOUL. Using ARABICA, queries are formulated using UML class diagrams and an UML profile that allow developers to describe which entities of the diagram are variable. Figure 4 illustrates the use of an UML class diagram to specify a query. The query expresses violations to a design contract that states that subclasses of the `AbstractTool` class which implement the `activate()` method must also invoke `super.activate()`. In order to specify this design contract violation, two classes are defined: `AbstractTool` and `?subClass`. The former is a variable class, specified by the `Variable` stereotype, which results on a *`?subClass`* variable included in the query. An `extends` relation between the two classes is marked with the stereotype `Transitive` to specify that classes bound to *`?subClass`* must (indirectly) inherit from `AbstractTool`. Finally, and operation `activate` is added to the `?subClass` class. This operation is marked with a stereotype `Templated`. This stereotype contains an attribute that links it to an UML comment stereotyped with `Template` to describe the characteristics that the operation must exhibit: not contain a call to `super.activate()`.

The ARABICA UML profile defines the four stereotypes: «Variable» named elements for entities whose names are mapped to SOUL variables, «Transitive» relations, «Templated» operations and comments that specify operation «Templates».

The ARABICA plugin extends the BARISTA-UI editor by embedding an UML editor in place of the query editor. All other features offered by the BARISTA-UI plugin remain unchanged. In addition to extending the BARISTA-UI plugin, ARABICA reuses the TopCased UML editor [2], and leverages Acceleo [1] to implement a model to text transformation that converts stereotyped UML diagrams into SOUL queries.

IV. SUMMARY AND TOOL AVAILABILITY

We have presented BARISTA, a tool suite to query programs using the logic program query language SOUL. BARISTA offers a comprehensive querying environment, as well as extension points to other tools that require code querying facilities. The BARISTA tool suite can be downloaded from http://soft.vub.ac.be/SOUL, where further information both on SOUL and the tool suite can be found.

ACKNOWLEDGEMENTS

This research is supported by the IAP Program of the Belgian State. Coen De Roover is funded by the *Stadium* SBO project sponsored by the "Flemish agency for Innovation by Science and Technology" (IWT Vlaanderen).

REFERENCES

[1] Acceleo MDA Model to Text Transformations, 2011. http://www.eclipse.org/acceleo/.
[2] TopCased Open-Source Toolkit for Critical Systems, 2011. http://www.topcased.org/.
[3] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 1(1), 1989.
[4] C. De Roover. *A Logic Meta Programming Foundation for Example-Driven Pattern Detection in Object-Oriented Programs*. PhD thesis, Vrije Universiteit Brussel, August 2009.
[5] C. De Roover, J. Brichau, C. Noguera, T. D'Hondt, and L. Duchien. Behavioral similarity matching using concrete source code templates in logic queries. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantics-based Program Manipulation (PEPM07)*, 2007.
[6] 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.
[7] K. Gybels, R. Wuyts, S. Ducasse, and M. D'Hondt. Inter-language reflection: A conceptual model and its implementation. *Elesevier International Journal on Computer Languages, Systems and Structures*, 32, 2006.
[8] 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.