The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse

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

Software Languages Lab, Vrije Universiteit Brussel, Belgium

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

Abstract

Program queries can answer important software engineering questions that range from *"which expressions are cast to this type?"* over *"does my program attempt to read from a closed file?"* to *"does my code follow the prescribed design?"*. In this paper, we present a comprehensive tool suite for querying Java programs. It consists of the logic program query language SOUL, the CAVA library of predicates for quantifying over an Eclipse workspace and the Eclipse plugin BARISTA for launching queries and inspecting their results. BARISTA allows other Eclipse plugins to peruse program query results which is facilitated by the symbiosis of SOUL with Java – setting SOUL apart from other program query languages. This symbiosis enables the CAVA library to forego the predominant transcription to logic facts of the queried program. Instead, the library queries the actual AST nodes used by Eclipse itself, making it trivial for any Eclipse plugin to find the AST nodes that correspond to a query result. Moreover, such plugins do not have to worry about having queried stale program information. We illustrate the extensibility of our suite by implementing a tool for co-evolving source code and annotations using program queries.

Categories and Subject Descriptors D.2.3 [*Coding Tools and Techniques*]: Object-Oriented Programming; D.2.6 [*Programming Environments*]: Eclipse; D.3.2 [*Language Classifications*]: Constraint and logic languages

General Terms Languages

Keywords program queries, logic programming, program analysis, software engineering tools, integrated development environments

1. 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 the literature (e.g., Prolog [13] and Datalog [4] have lent their proof procedure to [1, 7, 10, 15, 20, 26, 30] and [6, 12, 17, 22] respectively). We will focus on querying Java programs using the latest incarnation of SOUL [30], one of the earliest logic program query languages under active development.

User-specified program queries are of special interest. They enable enforcing coding conventions a development team has agreed upon, detecting violations against the protocol an in-house API expects to be adhered to, ensuring that an application's design rules are adhered to, or even checking whether there are similar instances of suboptimal code upon its discovery. As such, they provide application-specific support for the development process. Despite their valuable applications, logic program query languages have not yet become an integral part of every developer's toolbox. In our experience, this can be attributed to two adoption hurdles:

H1: Specifying program queries First of all, considerable expertise is required to specify source code characteristics through logic formulas that quantify over a reified program representation. Users have to be familiar with the representation of the queried program (e.g., the abstract grammar of AST nodes) as well as its reification (e.g., a transcription to logic terms). Moreover, such formulas tend to become convoluted and operational in nature when users attempt to ensure that all implementation variants of a characteristic are recalled. This is especially true for control flow (i.e., those that concern the order in which instructions are executed) and data flow (i.e., those that concern the values operated upon by instructions) characteristics.

H2: Exploiting solutions to program queries The second adoption hurdle constitutes the practical challenges that tool builders face in exploiting the results of a logic program query. Consider implementing a plugin 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. Not wanting to reinvent the wheel, we could resort either to invoking the search API of the IDE or to launching a logic program query. The latter is the only feasible option in case the sought after AST nodes are characterized behaviorally. Existing search APIs do not support control flow and data flow characteristics. Moreover, this option has the advantage of opening the tool up to potential application-specific uses without exposing users to its implementation details. We merely have to allow our users to modify the logic program query. However, finding the AST nodes that correspond to the solutions to this query can be more cumbersome. Solutions to a logic program query comprise reified program elements that are bound to the variables in the query. How cumbersome it is to find the AST node that corresponds to such a variable binding depends on the program representation and its reification.

PPPJ '11, August 24–26, 2011, Kongens Lyngby, Denmark.

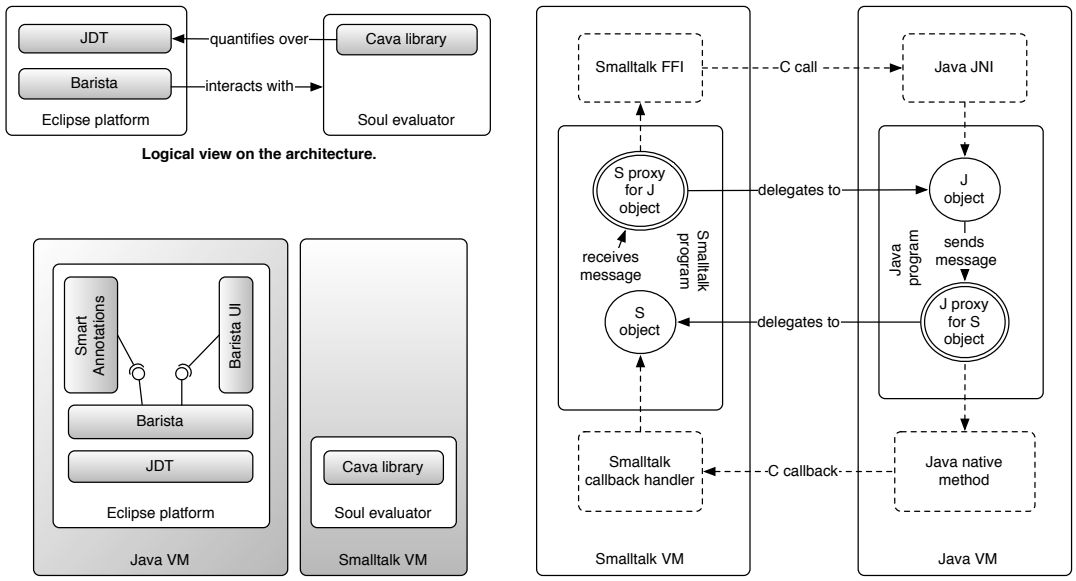


Figure 1. Architectural overview of the SOUL tool suite for querying Java programs.

To address the first adoption hurdle, SOUL has pioneered several features that facilitate querying Java programs. Among others, its example-driven matching [8] of code templates [9] recalls implementation variants of control flow and data flow characteristics —of which the code template exemplifies their prototypical implementation. To this end, SOUL consults the results of program analyses in its domain-specific unification procedure [3].

SOUL has featured a linguistic symbiosis [16] with Smalltalk since its inception. This symbiosis renders Smalltalk objects first-class values in the program query language. In addition, it allows evaluating arbitrary Smalltalk expressions in a program query. As a result, SOUL queries can quantify over any object that is reachable in a live Smalltalk image. More recently, we established a symbiosis between SOUL and Java through the Smalltalk-to-Java interconnection facilities provided by the JAVACONNECT library [2]. As a result, SOUL queries can quantify over any object that is part of a live Eclipse instance.

In this paper, we demonstrate how this SOUL-Java symbiosis facilitates exploiting the solutions to a logic program query in software engineering tools and Eclipse plugins in particular — thus addressing the second hurdle to the adoption of logic program querying. Among others, the symbiosis of SOUL and Java enables an identity-based reification of AST nodes. The reified version an AST node is the AST node itself (i.e., an instance of `org.eclipse.jdt.core.dom.ASTNode`). It is therefore trivial for any Eclipse plugin to find the AST nodes that correspond to a query result. Moreover, such plugins do not have to worry about having queried a stale program representation.

The contributions of this paper are as follows:

- An overview of the recent features of SOUL for querying Java programs (i.e., its predicate library, the example-driven matching of code templates and its domain-specific extensions to the unification procedure) and a discussion of how they facilitate specifying program queries (cf. adoption hurdle H1).
- We make the case for reifying AST nodes in logic program querying through linguistic symbiosis rather than the predominant transcription to logic facts. This facilitates exploiting the solutions to program queries in other software engineering tools (cf. adoption hurdle H2).
- The BARISTA plugin for Eclipse which provides extension points that other plugins can hook into for launching a SOUL query and exploiting its solutions. By means of a comparison of two re-implementations of our SMART ANNOTATIONS tool [19], we validate the advantages that logic program querying brings to tool builders. The first one relies on SOUL and BARISTA for identifying code, while the second one relies on the Eclipse search API instead.

2. Architecture of the Tool Suite

We begin our exposition with an overview of the architecture of the SOUL tool suite for querying Java programs in symbiosis with Eclipse. As depicted in the top-left corner of Figure 1, the tool suite consists of SOUL evaluator, the CAVA library of predicates for quantifying over Eclipse JDT projects and the Eclipse plugin BARISTA which provides extension points that other plugins can hook into for interacting with the SOUL evaluator. We will detail SOUL, the CAVA library and the BARISTA plugin in Section 3, Section 4 and Section 5 respectively. The bottom-left corner of Figure 1 depicts a physical view on the architecture. As the SOUL evaluator is implemented in Smalltalk, it runs on the Smalltalk VM. The Eclipse platform, on the other hand, runs on a Java VM that is started from within the Smalltalk VM. Terminating the Smalltalk VM also terminates the Java VM. Our BARISTA plugin runs on the Eclipse platform alongside the Eclipse JDT plugin that provides the IDE's facilities for editing Java programs. Plugins BARISTAUI and SMART ANNOTATIONS interact with the SOUL evaluator by hooking into the extension points of the BARISTA plugin. Both of these plugins will be discussed in sections 5.3 and 6.

***Overview of* JAVACONNECT** Central to the architecture of our tool suite are the Smalltalk-to-Java interconnection facilities provided by the JAVACONNECT [2] library. These enable invoking methods on Java objects (e.g., those that make up the Eclipse IDE) from within a Smalltalk image. This is illustrated by the following

snippet. It prints to the Smalltalk "Transcript" (i.e., a globally accessible log) the individual fragments of the class path that a particular Eclipse project p should be launched with:

```
1 a := JavaWorld.org.eclipse.jdt.launching.JavaRuntime
2       computeDefaultRuntimeClassPath_IJavaProject: p.
3 c := a asSmalltalkValue.
4 c do: [:s | Transcript write: s asSmalltalkValue]
```

Lines 1-2 invoke the static method `JavaRuntime.computeDefaultRuntimeClassPath(IJavaProject)` which returns a Java array of Strings that correspond to the fragments of the class path. The method is invoked through a Smalltalk selector `computeDefaultRuntimeClassPath_IJavaProject:`. JAVACONNECT automatically generates such Smalltalk selectors for Java method signatures. The Smalltalk selector consists of keywords of which the first is the concatenation of the Java method name and the short name of the type of the first argument. The remaining keywords are the short names of the types of the remaining arguments of the Java method.[1] Such selectors are understood by the Smalltalk proxies that JAVACONNECT creates for Java objects. The above snippet assumes that variable `p` is already bound to a Smalltalk proxy that JAVACONNECT created for a Java `IProject`. When line 2 passes this proxy as the argument to message `computeDefaultRuntimeClassPath_IJavaProject:`, JAVACONNECT invokes the corresponding Java method with the unwrapped `IProject`. It then creates a new Smalltalk proxy for the Java array that is returned by this method, which gets assigned to variable `a`.

One of those proxies is depicted as *"S proxy for J object"* on the right-hand side of Figure 1. It delegates messages that correspond to a Java method signature to the actual Java object the proxy was created for. To this end, JAVACONNECT calls one of the C procedures in the Java Native Interface (JNI) of the Java VM through the foreign function interface (FFI) of the Smalltalk VM. JAVACONNECT supports callbacks by dynamically generating a Java proxy for any Smalltalk object that is passed as an argument to a Java method. Methods invoked on such a Java proxy (e.g., *"J proxy for S object"* on the right-hand side of Figure 1) are delegated to the original Smalltalk object. To this end, the invocation handler of the generated Java proxy calls back into the Smalltalk VM through a Java Native Method.

JAVACONNECT supports decorating Smalltalk proxies for Java objects with plain Smalltalk methods. A number of convenience methods is predefined on proxies for commonly used Java objects. For instance, all Smalltalk proxies for instances of `java.lang.String` are decorated with an `asSmalltalkValue` method that returns a corresponding instance of Smalltalk's `ByteString` class. Likewise, all Smalltalk proxies for Java arrays answer message `asSmalltalkValue` with an equivalent instance of Smalltalk's `Array` class. The elements within the array are left unchanged. Line 3 of the above snippet therefore assigns variable `c` to a Smalltalk array that contains Smalltalk proxies for Java `Strings`. Line 4 iterates over them using the Smalltalk iteration message `do:`. This message is given a closure as its argument, which gets applied to each of the `Strings` `s` in the array. Through another invocation of `asSmalltalkValue`, each Java `String` is converted to an equivalent Smalltalk `ByteString` that is written to the `Transcript`.

Note that the users of our tool suite do not have to implement such Smalltalk snippets themselves. Eclipse users that want to identify source code of interest in a JDT project can do so declaratively through SOUL queries (cf. Section 3). Eclipse plugin builders that want to exploit the solutions to a SOUL query can do so by hooking into the Java extension points provided by the BARISTA plugin (cf. Section 5). Only the most basic predicates from our CAVA library are implemented in this manner (cf. Section 4).

[1] Fully qualified type names are used instead to disambiguate the Smalltalk selectors for a method that is overloaded on types that share the same short name.

3. The SOUL Program Query Language

SOUL is a logic-based program query language. 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. 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.

3.1 Syntax and Semantics in a Nutshell

The first two program queries depicted in Figure 2 illustrate the core syntax and semantics of SOUL. The query on line 1 identifies, among the AST nodes of an Eclipse JDT project, all pairs of Java statements `?inner` and `?outer` such that the former resides at an arbitrary depth within the latter. Such a pair of bindings for variables `?outer` and `?inner` constitutes a single solution to the query. The query consists of two conditions. The first condition uses unary predicate `isStatement/1` to bind `?outer` to one of the statements in the JDT project. The second condition of the query uses binary predicate `isStatementIn:/2` to bind `?inner` to one of the statements within the binding for `?outer`.

Note that SOUL queries start with the keyword `if` and that logic variables are preceded by a question mark. The syntax for a predicate in SOUL closely resembles the one of Smalltalk for a message that is sent to the first argument of the predicate. In Prolog, the second condition of the query would therefore be written as `isStatementIn(Inner,Outer)`.

Like Prolog, SOUL finds solutions to a logic query using SLD-resolution [13]. In case a condition can be proven in multiple ways (e.g., each resulting in different bindings for its variables), this procedure selects one proof but remembers those that have not yet been explored. When a subsequent condition fails, the proof procedure will backtrack to the last choice point and explore one of the remembered alternatives. For instance, the second condition of the query fails whenever `?outer` is bound to a statement in which no other statements reside. The first condition will then be backtracked to such that an alternative binding can be considered. This also happens when the SOUL evaluator is asked for the next or all solutions to a query.

The second query in Figure 2 is similar to the first one. Its solutions consist of an AST root node `?cu` and an AST node `?node` such that the latter is one of the children of the former. Using predicate `isCompilationUnit/1`, the first condition binds `?cu` to the root node of one of the ASTs for a JDT project (i.e., a compilation unit is the root node of the AST for a .java file). When this condition is backtracked over, `?cu` will be bound to a different compilation unit of the project. The second condition uses predicate `isChildOf:/2` to bind `?node` to a child of `?cu`. When this condition is backtracked over, `?node` will be bound to one of the other children until `?cu` has been traversed completely. As a result, the query quantifies over all compilation units and their children. All of the aforementioned predicates stem from the CAVA library which is discussed in Section 4.

3.2 Features that Facilitate Exploiting Query Solutions

In addition to the logic and its proof procedure, the program representation and its reification comprise two important dimensions in

```
1  if ?outer isStatement, ?inner isStatementIn: ?outer

2  if ?cu isCompilationUnit, ?node isChildOf: ?cu

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

7  if ?return isStatement,
8     ?return returnStatementHasExpression: ?exp,
9     ?exp equals: castExpression(?type, ?casted)

10 if jtStatement(?return){ return (?type) ?casted; }
```

```
11 if jtClassDeclaration(?classDeclaration) {
12       class ?className {
13         private ?fieldDeclarationType ?fieldName;
14         ?modifierList ?returnType ?methodName(?parameterList) {
15            return ?fieldName;
16         }
17       }
18    }

19 if jtMethodDeclaration(?m){
20      public static void main(String[] args) {
21        ?scanner := [new java.util.Scanner(?argList);]
22        ?scanner.close();
23        ?scanner.next();
24      }
25    }
```

Figure 2. SOUL queries illustrating the use of logic predicates (first and second query), object-oriented expressions (third query), code templates (the fourth and fifth queries) and the example-driven matching of code templates (the last two queries on the right-hand side).

the design of a logic-based program query language. In our experience, representation and reification determine how cumbersome it is for a tool builder to exploit the solutions to a program query in a software engineering tool.

To illustrate the impact of representation and reification in the use of a logic program query language, consider an abstract syntax tree with root `a`. Root node `a` has two children `b` and `c`, while node `c` has a child `d`. Now suppose a logic variable is bound to the reification of `d` and that we have to find `d` itself. This entails traversing the reification of `a` and `a` itself simultaneously until we reach the reification of `d` and `d` itself respectively. This is already less straightforward for a reification that flattens the AST into multiple logic terms (e.g., `child(a,b)`, `child(a,c)` and `child(c,d)`) than for a reification that maps the AST to a single compound term (e.g., `node(a,<node(b,<>),node(c,<node(d,<>)>))`). However, the logic program query language and the software engineering tool need not use the same program representation. 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. They often use control flow graphs or the results of data flow analyses. 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).

Our approach to overcoming this adoption hurdle is two-fold:

- First of all, the predicates of the CAVA library (cf. Section 4) only reify AST nodes. While some predicates consult program analyses that are computed by the SOOT framework [27] for its JIMPLE intermediate representation, JIMPLE elements cannot 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, the hybrid language characteristic (or more precisely, its symbiosis [16] with Smalltalk) of SOUL 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* (i.e., an instance of `org.eclipse.jdt.core.dom.ASTNode`).

The latter is illustrated by the query on lines 3–6 of Figure 2. Its solutions consist of bindings for variables *?m*, *?i* and *?t* such that *?m* is a method declaration (line 3) as well as the *?i*-th member declaration of the type declaration *?t* (lines 4–6). The condition on line 4 uses predicate `equals:/2` to unify[2] variable *?t* with a so-called *Smalltalk term*. *Smalltalk terms* are delimited by square brackets and can contain logic variables wherever Smalltalk variables are allowed. They can quantify over any object that is reachable in the Smalltalk image at run-time —including the Smalltalk proxies created by JAVACONNECT for Java objects (cf. Section 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. After logic variables have been substituted by the values they are bound to, the expression within a *Smalltalk term* is evaluated as regular Smalltalk code. In the case of the Smalltalk term on line 4, *?t* will be bound to the parent node of the method declaration AST node bound to *?m*. This is a type declaration (e.g., an interface or class) of which the list of declarations in its body is bound to *?l* by the condition on line 5. Note that this is not a logic list, but an instance of `ASTNode$NodeList`. As illustrated by line 6, it is therefore possible to ask this list *?l* for the index of *?m* within the body of the type declaration. Were AST nodes reified as compound terms, it would not have been possible to query them for their context within the program through method invocations.

Note that *Smalltalk terms* are rarely used directly in queries over Java programs. Indeed, they expose SOUL users to the way JAVACONNECT maps Smalltalk to Java (cf. Section 2) and to the internals of the Eclipse JDT. Instead, such terms are hidden in the implementation of the CAVA predicates (cf. Section 4) that reify the AST nodes of a JDT project. Relying on linguistic symbiosis for the reification renders reconstituting AST nodes from their reified counterparts trivial —thus facilitating the manipulation of query results in software engineering tools.

3.3 Features that Facilitate Specifying Program Queries

In this section, we discuss features pioneered by SOUL that facilitate specifying program queries: domain-specific extensions [8] to an open unification procedure [3], and an example-driven matching [8] of code templates [9]. Together, they enable *exemplifying* source code characteristics through familiar code templates rather than specifying them through convoluted logic conditions. For instance, the query on lines 7–9 of Figure 2 and the one on line 10 are equivalent. Both identify statements *?return* that return the result of *?casted* being cast to *?type*.

A template term consists of a functor (e.g., `jtStatement` on line 10) followed by an argument (e.g., *?return* on line 10) and a code excerpt that is demarcated by braces. The functor of the

[2] The `equals:/2` predicate is implemented by the fact *?x* `equals:` *?x*. It is equivalent to the `=/2` operator in Prolog.

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. Within a code excerpt, logic variables stand for productions that originate from a non-terminal in the Java grammar.

Example-driven matching of template terms Used as a condition, a template term succeeds if there is an AST node that matches the code excerpt. Backtracking over the term successively unifies each matching node with the argument of the term. We match such excerpts against the program's ASTs according to multiple matching strategies that vary in leniency —all of which are considered successively when the term is backtracked over[3].

The *most stringent matching strategy* requires the AST node to unify with the AST of the code excerpt. It corresponds to the one used by most query languages that incorporate code excerpts. This strategy suffices to find real-life matches for the template term on line 10, but not for those on the right-hand side of Figure 2. The first and second template term on the right exemplify the prototypical implementation of a getter method (i.e., a best practice pattern) and a `Scanner` that is read from after it has been closed (i.e., a bug pattern) respectively. More lenient matching strategies are needed to find real-life variants of these implementations in a program's source code. We refer to these strategies as example-driven matching strategies [8].

To illustrate the *most lenient matching strategy*, consider the second template term. On line 21, it uses one of the aforementioned non-native syntax extensions. The `:=` operator unifies the logic variable on its left-hand side (i.e., *?scanner*) with the AST node that matches the code template on its right-hand side (i.e., an instance creation expression). On the next lines, the occurrences of *?scanner* ensure that `close()` and `next()` are invoked on the same receiver. The actual matching strategy only requires matching methods to exhibit the *control flow* characteristics exemplified by this template term. In other words, there should be a path through the control flow graph of the method (i.e., existentially qualified) on which all exemplified instructions are executed in the exemplified order (i.e., `close()` before `next()`). Non-specified instructions are allowed on the execution path. The path is also allowed to cross method boundaries (i.e., it is inter-procedural). As a result, matches for an instruction in the template term need not reside in the method declaration that matches the term.[4] While *?m* will always be bound to `main(String[])`, the `close()` and `next()` instructions can reside in methods called directly or transitively from `main(String[])`.

Domain-specific unification SOUL has also pioneered extensions [8] to its open-ended unification procedure [3] that are specific to querying Java programs. These unification extensions treat reified AST nodes different from other logic terms:

- Reified AST nodes unify with structurally equivalent compound terms, even if they have not been reified as such. This is the case for our reification through linguistic symbiosis (cf. Section 3.2). While this reification facilitates exploiting query results, it precludes the natural use of unification to quantify selectively over AST nodes.
- Implicit implementation variants (i.e., those implied by the semantics of the programming language) of the same source code characteristic unify. This relieves users from having to enumerate these variants in a specification. To this end, these unification extensions consult the results of analyses computed by the SOOT framework [27]. For instance, a semantic analysis determines whether an unqualified and fully qualified type should unify. An alias analysis [18] determines whether two expressions should unify. The names of a method invocation and a method declaration unify if the invocation may invoke the declaration. We refer to [8] for a comprehensive account of these extensions.

To illustrate the first extension, consider the condition on line 9 of Figure 2. It requires AST node *?exp* to unify with a structurally equivalent compound term (i.e., the functor of the compound has to unify with the name of the node's class, while each of its arguments has to unify with the corresponding child of the node). This would fail under the regular unification procedure of Prolog as *?exp* is bound to a reified AST node (i.e., an object).

To illustrate the second kind of extensions, consider the query on lines 11–18 of Figure 2. Its occurrences of *?fieldName* link the value returned by the getter method to the field it is protecting. Under the domain-specific unification procedure, the bindings for these variable occurrences can deviate syntactically as long as they are in an alias relation. This ensures that getter methods with a deviating `return`-statement (e.g., "`return this.age`" or even "`return (Integer) self().age`") are included in the solutions to the query. In the query on lines 19–25, the occurrences of *?scanner* require the receiver of `close()` and `next()` to alias (i.e., point to the same `Scanner` instance).

4. CAVA: a Predicate Library for SOUL

The CAVA library contains SOUL clauses that implement predicates for reasoning about Java programs. For each subclass of `org.eclipse.jdt.core.dom.ASTNode`, the CAVA library provides a unary predicate that quantifies over all AST nodes of this kind. For instance, predicate `isMethodDeclaration/1` can be used to bind its argument to a method declaration, but also to check whether its argument is already bound to a method declaration.

Binary predicates quantify over the relations between an AST node and its children. For instance, the condition on line 5 of Figure 2 uses predicate `typeDeclarationHasBodyDeclaration:/2` to bind (or verify the binding of) variable *?l* to the list of body declarations in a type declaration *?t*. In addition, CAVA provides higher-level predicates that quantify over relations between AST nodes that are not explicit in the AST representation. Examples include predicate `isStatementIn:/2` which quantifies over all statements that reside at an arbitrary depth within another statement (cf. line 1 of Figure 2), predicate `inheritsFrom:/2` which quantifies over the inheritance relation between types, or predicate `isChildOf:/2` which quantifies over the relation between a node and its descendants (cf. line 2 of of Figure 2).

A more comprehensive account of the CAVA predicates and their implementation can be found in [8].

5. BARISTA: Exploiting Query Solutions in Eclipse Plugins

BARISTA is an Eclipse plugin that serves as communication bridge between the IDE and SOUL. Its goal is to offer, by means of the Eclipse plugin extension mechanism, services to other plugins that require code querying facilities.

BARISTA offers two main services: on-demand code querying, and query scheduling. On-demand querying allows plugins depending on BARISTA to query any given Java project in the Eclipse workspace, the results of the query immediately returned back to the client plugin. Plugins depending on BARISTA can also schedule queries to be run whenever a given project is built by the Eclipse

[3] A matching strategy can also be specified as an additional argument to the template term.

[4] However, actual statements such as `return`-statements are matched intra-procedurally.

```
public interface IBarista {
 public IEvaluator query(String query,
                 IJavaProject project,
                     String evaluator,
                    String repository);
 public List<String> getEvaluators();
 public List<String> getRepositories();
 ... }

public interface IEvaluator {
 public IResults getAllResults();
 public IResults getNextResult();
 public IResults getNextResults(int amnt);
 public boolean hasMoreResults(); }

public interface IResults {
 public int getSize();
 public Map<String,List<Object>> toMap();
 public boolean isSuccess();
 public long getElapsedTime(); }
```

Figure 3. Interfaces provided by BARISTA for code querying and query scheduling.

JDT framework. This allows plugins to maintain up-to-date information on a project.

5.1 On-demand code querying

BARISTA offers the service of on-demand code querying to depending plugins via the `IBarista`, `IEvaluator` and `IResults` interfaces (figure 3), with `IBarista` being the main access point to the functionality offered by the plugin. Access to this interface is provided by a static method on the main class of the BARISTA plugin.

The main service offered by the `IBarista` interface is the `IBarista.query(...)` method. The method takes as parameters the query to run, on which project to run it, and what evaluator and clause repository to use. The query method prepares a query, returning an `IEvaluator` object that allows the client to get all results at once, or one at the time. Each result is encapsulated in an `IResult` object — a map between logic variables in the query and their bindings. In reality, the instances implementing the aforementioned interfaces are not Java objects but Smalltalk objects. The way in which this is achieved is explained in section 7.

5.2 Query Scheduling

BARISTA allows client plugins to schedule queries to be run at each build of the system through the exposure of an Eclipse plugin extension point. This is useful for clients performing queries that give feedback to the developer (e.g., identifying errors, design violations, calculating metrics, etc.) Client plugins that implement this extension point can then register queries, and will be notified with the results of the queries every time they are executed.

Whenever a client plugin requires a set of queries to be run when a project changes it must provide, by means of the `barista.scheduleQueries` extension point, a callback class that implements the `IQueryManager` interface. The `getQueries()` method is called by the BARISTA runtime at the beginning of a project build to gather the queries from all client plugins. Once the build is finished, the BARISTA runtime will run all schedule queries, and pass the results back to each query manager by means of the `resultsDone(...)` method.

5.3 BARISTAUI: Editors for Queries and Inspectors for their Solutions

Query Editor In order to ease the development of BARISTA-dependent plugins, 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. Through Barista-UI, plugin developers can test queries on projects within their same project. Each of the components of the Barista-UI maps to the interfaces provided by BARISTA (§ 5.1), thus easing the use of the UI in the development of plugins that extend BARISTA. The Barista-UI also serves as a general-purpose code querying facility, complementing that of the Eclipse JDT.

The Barista-UI plugin provides an advanced query editor (figure 4 on the left), complete with syntax-highlighting, auto-completion, syntax-error checking and contextual help. Using the query configuration pane the developer can select the project, evaluator and clause repository to execute the query. Notice that the query editor maps directly to the arguments of the `IBarista.query(...)` (figure 3) method explained in the previous section.

Inspecting Query Results Once the developer is satisfied with it, the query can be executed. The *Prepare Query* button on the editor will fire off a (new) *Query Results* view (figure 4 on the right). This view allows the developer to control how to obtain the solutions to the query: either all at once via the *All Results* button, or one at a time via the *Next Result* button. These controls correspond to the methods offered by the `IEvaluator` interface.

Barista-UI offers three visualization options for the results of a query: as a table, tree or column view. These views mirror the ones provided by SOUL in Smalltalk. Ordering of the variables can be changed under the `Query Variables` configuration pane at the top of the view. In addition to this, Barista-UI offers facilities to inspect individual bindings and mark the results of the query.

Inspecting Individual Results A context menu is available for each of the results on the query view. Through this menu, the developer can navigate to the position in the source code that corresponds to the binding, copy a string representation of the binding to the clipboard, or open a Smalltalk inspector on the binding. The context menu, as well as the inspector are depicted in figure 5a. This is made possible by the preservation of object identities when they pass from Java to Smalltalk and back (§ 7).

Mark Results By clicking on the *Mark Results* button on the query results view, the Barista-UI will mark all the results of the query which resolved to an Eclipse `ASTNode`. These marks will be displayed on the left side margin of Java editors. Figure 5b. shows one of these markers. Each marker contains information on the bindings for the other variables. In the example, this will allow the developer to from the marker of one variable, to navigate to the other bindings. This functionality is made possible by the fact that the results for every query are actual `ASTNodes`, which makes the use of Eclipse functionality to navigate to them rather easy.

6. SMART ANNOTATIONS: Building tools with BARISTA

In this section we present a reimplementation of SMART ANNOTATIONS [19]. SMART ANNOTATIONS allow developers to express constraints over the use of Java annotation as SOUL queries. These queries are used to assert the correctness of annotated program elements with regards to their annotations. SMART ANNOTATIONS illustrate the utility of SOUL, both as a specification language for developers, and as a code querying facility for tool builders. To exemplify the latter case we compare two possible implementations of SMART ANNOTATIONS: one that relies on the Eclipse search API, while the second one uses BARISTA and SOUL.

6.1 SMART ANNOTATIONS

Several programming languages include facilities to attach metadata to source code, such is the case with Java's annotations. Used for more than plain documentation, annotations are becoming an

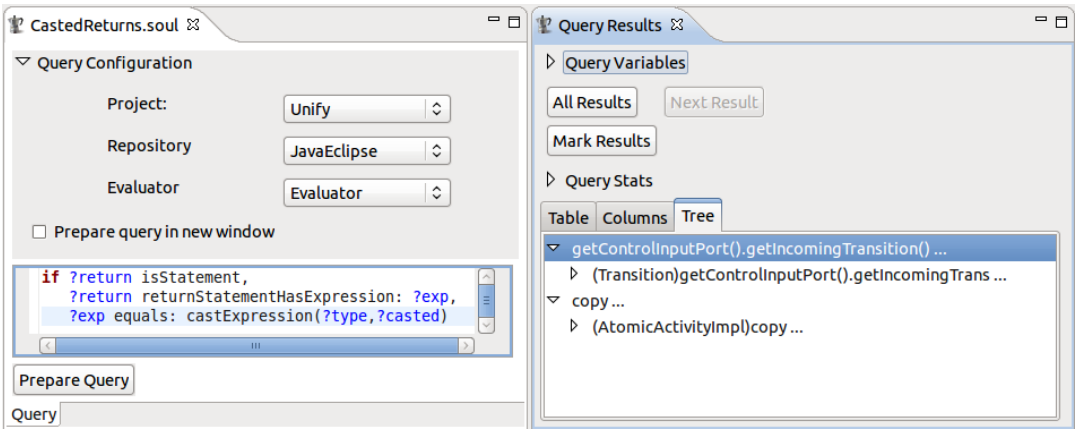
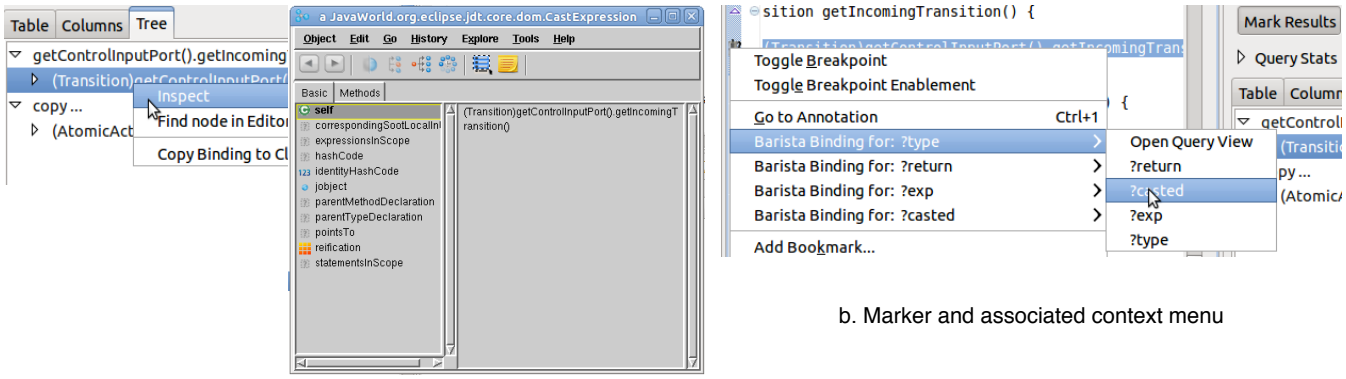


Figure 4. Query editor and query result views provided by the Barista-UI plugin.



a. Context menu and Inspector for a result

b. Marker and associated context menu

Figure 5. Markers on the results of a query.

integral part of the system. Frameworks provide annotations as a means of embedding configuration parameters in the source code. An example of this practice is found in the Hibernate persistency framework, where annotations are placed on the source code by the developer to direct how the framework will persist entities. Hibernate will then, at runtime or compilation time, read the annotated source code in order to derive in which tables and columns will particular classes and fields be persisted.

Developers must rely on documentation to assure that they are respecting the assumptions of the framework when annotating their code, or else risk errors at runtime when rules are broken. SMART ANNOTATIONS alleviate this problem by offering a means to check that annotated code respects the rules of use of annotations. SMART ANNOTATIONS enriches Java annotation type definitions with constraints expressed using SOUL that govern the correct use of these annotations. The tool allows these constraints to be verified with respect to the source code, will report wrongly annotated source-code entities or source-code entities where an annotation is missing, and provide a number of quick fixes resolve the errors.

Constraints are embedded in annotation-type declarations as string constants (public static final fields) that contain a SOUL query codifying each constraint. SMART ANNOTATIONS define two kinds of constraints: *necessary* constraints express the characteristics that annotated source code elements must exhibit, while *sufficient* constraints express the characteristics that source code elements must exhibit in order to be annotated. Fields containing annotation constraints must then be annotated with `@Necessary` or `@Sufficient` to specify the kind of constraint. Figure 6 shows the `Getter` annotation-type definition, which is meant to mark methods that act as accessors. A single necessary constraint (`NAMING_CONVENTION`) is defined in order to check that methods annotated as `@Getter` follow the naming convention.

```
1 public @interface Getter{
2   @Necessary
3   public static final String NAMING_CONVENTION =
4     "?method methodDeclarationHasName: {get*}";
5 }
```

Figure 6. Annotation-type with embedded SMART ANNOTATIONS constraints

6.2 Implementation

Checking that a system's source code respects the annotation's constraints as defined in SMART ANNOTATIONS is a two step process. First, the set of constraints, embedded in the annotation-type definitions must be gathered. Once obtained, the SOUL queries defining the constraints must be run, and their violations marked as errors. In its current incarnation, SMART ANNOTATIONS is implemented as an Eclipse plugin that leverages BARISTA for both these steps.

To check annotated source code, the new SMART ANNOTATIONS implementation will schedule the set of SOUL queries that identify violations the constraints defined over annotations in the

system. To extract these queries from the annotation-type definitions, the SMART ANNOTATIONS plugin uses the `IBarista` interface to launch a SOUL query that quantifies over the annotation type definitions and retrieves the constraints governing these annotations.

The gathered queries are then returned to BARISTA and scheduled for execution. When the scheduled queries are executed, BARISTA invokes the `resultsDone()` method on a `SmartAnnotationQueryManager` class. Once the scheduled queries representing the annotation constraints are executed by BARISTA, the SMART ANNOTATIONS plugin will retrieve the bindings for violations (these are actual `ASTNode` objects), and will create a warning marker for each of these violations.

6.3 Comparing Eclipse Search API and BARISTA

In order to illustrate the advantages that BARISTA and SOUL bring to tool builders, we compare how the first step (i.e., the gathering of annotation constraints) would be implemented using Eclipse's search API and using BARISTA and SOUL.

Gathering constraints with Eclipse Search API Eclipse, through the Java Development Tooling (JDT), provides developers with a search API for Java programs that offers other plugins search facilities to find source code element declarations and references. The JDT search API centers around three main concepts: *search patterns* specify the characteristics of the sought source code elements, *search scopes* restrict where the source code element will be found, and *search requestors* accumulate results provided by the API's search engine.

In the first step in the checking of annotation constraints, the SMART ANNOTATIONS plugin must find public static final fields of type String which belong to annotation-type declarations, and that are annotated with either `@Necessary` or `@Sufficient`. A search pattern expressing this constraint is rather difficult to express using the JDT's search API, since the API is geared towards expressing the characteristics of a single code entity (i.e., its type) and not relationships between entities (i.e., the field belongs to an annotation-type). Thus we opt for an over-approximating query that relies on the fact that we are searching for annotated items.

Figure 7 presents the snippet, that out of a Java project, finds the constraints, the field in which they are defined and the annotation-type to which they are associated. Lines 1–5 and 7–11 construct search patterns for `@Necessay` and `@Sufficient`-annotated elements. These patterns are composed into an *or* pattern `allConditions` in lines 13–14. The scope of the search is restricted to the project `jProject` in lines 16–18, and a custom requestor `SimpleRequestor` that will accumulate the annotations is constructed in line 20. Finally, the search is done in lines 24-27. Results gathered by the requestor are navigated in lines 29–37, using the `getAncestor()` method to obtain the field containing the constraint and the annotation-type associated with it.

***Gathering constraints using* BARISTA** Figure 8 contains the listing for gathering the SMART ANNOTATIONS constraints out of annotation types using BARISTA's interfaces. The query is performed in two steps. First, an evaluator (lines 1–11) is constructed. The evaluator is obtained from an `IBarista` instance (cf. 5.1), a String containing the query, and the project `javaProject` in which the query is to be run. The query itself (lines 3–10) will provide bindings for the annotation type (*`?type`*), the field that contains the constraint (*`?field`*) as well as the constraint itself (*`?rule`*). Results for the query are obtained by invoking `getAllResults()` on the evaluator, and extracting the map from variables to bindings (lines 13 and 15). A loop then iterates over the solutions, extracting the bindings for the constraint, kind of rule and annotation type. (lines 17–22).

```
1 SearchPattern allNecesary = SearchPattern.createPattern(
2   "be.ac.vub.smart_annotations_library.Necessary",
3   IJavaSearchConstants.ANNOTATION_TYPE,
4   IJavaSearchConstants.ANNOTATION_TYPE_REFERENCE,
5   SearchPattern.R_EXACT_MATCH);

7 SearchPattern allSufficient = SearchPattern.createPattern(
8  "be.ac.vub.smart_annotations_library.Sufficient",
9  IJavaSearchConstants.ANNOTATION_TYPE,
10 IJavaSearchConstants.ANNOTATION_TYPE_REFERENCE,
11 SearchPattern.R_EXACT_MATCH);

13 SearchPattern allConditions = SearchPattern.
14   createOrPattern(allNecesary, allSufficient);

16 IJavaSearchScope projectScope = SearchEngine
17  .createJavaSearchScope(
18  new IJavaElement[] { jProject }, false);

20 SimpleSearchRequestor requestor = new SimpleSearchRequestor();

22 SearchEngine engine = new SearchEngine();

24 engine.search(allConditions,
25  new SearchParticipant[]
26     {SearchEngine.getDefaultSearchParticipant() },
27  projectScope,requestor, null);

29 for (Object res : requestor.getResult()) {
30  IAnnotation ann = (IAnnotation) res;
31  IField field = (IField) ann.
32              getAncestor(IJavaModel.FIELD);
33  IType annotation = (IType) ann.
34              getAncestor(IJavaModel.TYPE);
35  String rule = (String) field.getConstant();
36  // handle constraint
37 }
```

Figure 7. Gathering constraints using Eclipse search API

```
1 IEvaluator eval = barista
2   .query(
3 "if ?type annotationTypeDeclarationHasName: ?annotation,"
4 +"?type definesVariable: ?field, "
5 +"?field variableDeclarationFragmentHasInitializer: ?rule,"
6 +"?field variableDeclarationFragmentHasName:  ?rulename,"
7 +"?field variableHasAnnotation: ? named:  ?annotationType,"
8 +"or(
9  equals(?annotationType,{Necessary}),
10  equals(?annotationType,{Sufficient}))",
11   javaProject, "Evaluator", "JavaEclipse");

13 IResults iresults = eval.getAllResults();

15 Map<String, List<Object>> results = iresults.toMap();

17 for (int i = 0; i < iresults.getSize(); i++) {
18  ASTNode annType = (ASTNode) results.get("?type").get(i);
19  String type = results.get("?annotationType").get(i).toString();
20  String ruleString = results.get("?rule").get(i).toString();
21  //Handle constraint
22 }
```

Figure 8. Gathering constraints using BARISTA

7. Discussion

Comparing* BARISTA *and Eclipse JDT Search In the previous section, we presented two possible ways of gathering information about Java source code in the context of the SMART ANNOTATIONS plugin: through the Eclipse JDT Search API and through our tool suite. We believe that the latter is not only less operational in nature, but also conceptually simpler. In the case of the Search API, the developer must deal with search patterns, scopes and requestors. She must decompose the query into a (number) of patterns, each one characterizing individual nodes, and then specify the scope of the query. Additionally, the developer has to implement a class that extends `SearchRequestor`, which will be called back by the Eclipse search engine for each matched element.

In contrast, the BARISTA interface to SOUL allows for a simplified interaction. Given a SOUL query and a Java project to evaluate it against, the developer is provided a map that contains the solutions to the query. Additionally, a single query can provide several results in each solution: the query in Figure 8 results in bindings for the annotation type, field name and constraint. The Eclipse search API, on the other hand, will only return a list of single matches. Reconstructing the relationships between the resulting source code elements is left up to the developer. For example, by navigating the

program's ASTs anew —the `getAncestor()` invocations in Figure 7).

When implementing an Eclipse plugin that searches through a JDT project, tool builders typically resort to implementing visitors over the ASTs of the project. Compared to our tool suite (and even the Eclipse search API), visitors seem harder to implement and maintain due to their imperative nature, proneness to code duplication, and the fact that the developer must keep track of the visited nodes in order to construct relations between them. Compared to a logic program query language, which enables developers to express the characteristics of the code they are looking for in a declarative manner, visitors require developers to implement the operational search for these nodes themselves.

Limitations of SLD-resolution Like Prolog, SOUL uses SLD-resolution [13] to find the solutions to a program query (cf. Section 3.1). This proof procedure has two well-known limitations: it does not terminate for certain kinds of recursive rules and it repeats sub-proofs for identical goals in queries or rules. The former limitation detracts from the declarative nature of logic programming (i.e., declaratively sound programs might not terminate operationally), while the latter limitation poses a burden on efficiency. In practice, most SOUL users will never be confronted with non-termination. It only arises for user-defined logic predicates. For instance, when a user implements a recursive graph traversal predicate without taking cycles into account.

SLG-resolution [5] overcomes both limitations of SLD-resolution through a clever tabling and delaying of sub-proofs. Although SOUL uses SLD-resolution by default, predicates that are implemented by a tabled rule (i.e., one annotated with the `tabled` keyword) will be resolved using SLG-resolution instead. To this end, the SOUL preprocessor transforms a tabled rule to a set of plain rules according to a transformation by Ramesh and Chen [25]. In our implementation, the resulting rules rely on *Smalltalk terms* (cf. Section 3.2) to access the SOUL evaluator and its tabling-related data structures. In the interest of a possible performance gain, we are currently in the process of annotating the implementation of frequently used predicates form the CAVA library (cf. Section 4) as `tabled`. The downside to the use of tabling is increased memory usage and the need to maintain tables as the queried program changes.

Technical Limitations In terms of deployment, the SOUL tool suite is currently limited by its architecture (cf. Section 2) that requires launching a Java VM from within the Smalltalk VM that executes the SOUL evaluator. This limitation stems from the lack of a C interface to the VisualWorks Smalltalk VM that allows launching the Smalltalk VM programmatically —in essence a native interface to the VM akin to the JNI. We are actively looking for a solution that would allow us to launch, from within the BARISTA Eclipse plugin, a Smalltalk VM that houses the SOUL engine. This way, we would be able to distribute our tool suite as a self-contained Eclipse plugin. As an intermediate solution, our distribution currently includes a headless Smalltalk image that starts a Java VM and an instance of Eclipse upon its launch.

One might argue that we could have avoided this technical limitation by developing in Java in the first place. This was not a viable option given the effort that would be required to reach a level of maturity and functionality that is equivalent to the existing code base of SOUL.There are also practical advantages to our approach. As JAVACONNECT supports decorating proxies for Java objects with Smalltalk methods and instance variables (cf. Section 2), we are able to extend the functionality of the Eclipse AST nodes without altering them. Among others, we implemented the domain-specific extensions to the unification procedure in this manner.

8. Related Work

In previous work [9], we already explored the idea of matching of code templates in a non-strict manner. However, the single matching strategy presented there aimed at recalling implementation variants of structural and data flow characteristics only. The example-driven matching of code templates presented in Section 3.3 employs multiple matching strategies that recall implementation variants of structural, control flow and data flow characteristics. We also explored the idea of an open-ended unification procedure for logic program query languages in previous work [3]. Concretely, we presented a procedure that unifies syntactically differing, but aliasing expressions. The extensions to this procedure presented in Section 3.3 unify additional implementation variants of source code characteristics (e.g., the name of a method invocation unifies with the name of the invoked declaration).

Completely new to this paper is the case for reifying AST nodes through linguistic symbiosis, rather than the predominant transcription to logic facts. The same goes for the BARISTA Eclipse plugin and its validation through a re-implementation of the SMART ANNOTATIONS plugin. As a result, this paper provides a comprehensive overview of the SOUL tool suite for querying Eclipse JDT projects.

A wide range of specification languages for source code characteristics has been proposed. Specification languages for structural characteristics include graph rewrite rules [24] or logic formulas [1, 6, 7, 10, 17, 26, 30] that quantify or range over a program's AST nodes. Specification languages for behavioral characteristics include reachability queries [11, 28, 29], temporal logic formulas [21], state machines [14] and logic formulas [15, 22] that quantify over control flow and data flow analysis results.

In contrast to these dedicated and highly specialized specification languages, SOUL supports specifying structural as well as behavioral characteristics in a uniform language that is familiar to developers. Support for code excerpts is not uncommon in specification languages (e.g., [1, 14, 22, 23, 26, 29]). However, it is usually restricted to individual expressions, statements and declarations. Larger excerpts would have to be matched in a one-to-one manner. To the best of our knowledge, SOUL's example-driven matching semantics for code excerpts is therefore unique.

Of the logic-based query languages, ASTLOG [7] is closest to SOUL with respect to the reification of AST nodes. Although it foregoes a transcription to compound terms, this comes at the cost of requiring predicates to be evaluated against the nodes that are encountered during an AST traversal. This exposes developers to the operational nature of these traversals.

JQUERY [10] is an interesting program navigation plug-in for Eclipse that uses the logic programming language TYRUBA to quantify over the workspace *and* to configure its graphical interface. We consider this kind of configurability an interesting venue for future work. The same goes for the incremental proof procedure, based on tabled resolution (cf. Section 7), that Eichberg et al. [12] have adopted for their logic program query language. It avoids reconsidering the entire search space for a proof upon a change to the program's code. This way, queries can be launched alongside the build process of the IDE without incurring a performance overhead.

9. Summary

In this paper we have presented a comprehensive tool suite for querying Eclipse JDT projects. Its main components are the logic program query language SOUL and the BARISTA plugin for interacting with the SOUL evaluator. We argued how the former's symbiosis with Java enables an identity-based reification of AST nodes that facilitates exploiting the solutions to a SOUL query in

software engineering tools. The latter provides an extensible architecture through which tool builders can access this powerful query language in which structural, control flow and data flow characteristics of source code can be specified in an example-driven manner.

BARISTAUI and SMART ANNOTATIONS comprise two examples of query-based tools that leverage the features of our tool-suite. The former serves both as an example of how to extend BARISTA and as a tool that is useful in the development of SOUL queries, while the latter illustrates how a code checking tool can be constructed easily by relying on the services exported by BARISTA and the expressive power of SOUL.

Availability and Access

SOUL and its associated tools have been developed under the MIT License. We refer to `http://soft.vub.ac.be/SOUL` for more information and download instructions.

Acknowledgements

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

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] J. Brichau and C. De Roover. Language-shifting objects in inter-language interoperability. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST09)*, 2009.

[3] J. Brichau, C. De Roover, and K. Mens. Open unification for program query languages. In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*, 2007.

[4] 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.

[5] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM (JACM). ACM*, 43(1):20–74, 1996. ISSN 0004-5411.

[6] 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.

[7] 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.

[8] C. De Roover. *A Logic Meta Programming Foundation for Example-Driven Pattern Detection in Object-Oriented Programs*. PhD thesis, Vrije Universiteit Brussel, August 2009.

[9] 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.

[10] 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.

[11] 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.

[12] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th International Conference on Software Engineering (ICSE08)*, 2008.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE01)*, 2001.

[19] A. Kellens, C. Noguera, K. De Schutter, C. De Roover, and T. D'Hondt. Co-evolving annotations and source code through smart annotations. In *Proceedings of the 14th European Conference on Software Maintenance and Re-engineering (CSMR10)*, 2010.

[20] K. Klose and K. Ostermann. Modular logic metaprogramming. In *Proceedings of the ACM International Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA10)*, 2010.

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

[22] 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.

[23] M. Mossienko. Structural search and replace: What, why, and how-to. JetBrains onBoard Online Magazine, February 2005. `http://www.onboard.jetbrains.com/is1/articles/04/10/ssr/`.

[24] 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.

[25] R. Ramesh and W. Chen. Implementation of tabled evaluation with delaying in Prolog. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):559–574, 1997. ISSN 1041-4347.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.