Example-based Program Querying

Andy Kellens
Vrije Universiteit Brussel
Brussels, Belgium
andy.kellens@vub.ac.be

Johan Brichau
Université catholique de Louvain
Louvain-la-Neuve, Belgium
johan.brichau@uclouvain.be

Coen De Roover
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.ac.be

Abstract

Program query languages are an essential component of program analysis and manipulation systems. In each such system, a query identifies the source-code parts of interest by reasoning over a program representation that is dedicated to the intent of the system (e.g. call-graphs to detect behavioral flaws, abstract syntax trees for refactorings, concrete source code to verify programming conventions, etc.). In order to detect a wide variety of such "patterns of interest", or more importantly, to detect patterns that require a combination of such program representations, developers must understand all the different applicable representations and techniques. We therefore present a logic-based language that allows the program's implementation to be queried using concrete source code templates that are matched against a combination of structural and behavioral program representations. These representations include call-graphs, points-to analysis results and abstract syntax trees that are uniformly composed through a customizable unification procedure. The result of our approach is that developers can detect patterns in the queried program using source code excerpts (embedded in logic queries) which act as prototypical samples of the structure and behavior they intend to match.

1. Introduction

The growing amount of program query languages —of which ASTLog [4], SOUL [16], JQuery [9], CodeQuest [8] and PQL [12] are only some examples— is testament to the significant momentum on the investigation of a program's structure and/or behavior by means of user-defined queries. Such queries serve the identification of code exhibiting features of interest which range from application-specific coding conventions [13] over metrics [10], refactoring opportunities [15] and design patterns [7] to run-time errors [12, 5].

A large body of these query languages are *logic* program query languages, meaning that they rely on the use of an executable logic to query the program under investigation. The use of a logic programming language to query programs has several well-established advantages [3, 16]. In imperative programming languages, programmers specify exactly *how* the solution to a problem is to be found using step-by-step algorithms. In contrast, logic programming languages allow the problem itself to be specified. The program will find a solution on its own, relying on a specific problem-solving strategy defined by the language. In such an approach, program queries are expressed as logic conditions over the program's parts.

In order for the aforementioned logic problem-solving strategy to work, the program under investigation is reified in a representation that is fine-tuned for a particular purpose. Systems that focus on the verification of coding conventions and styles generally only require a structural representation (i.e. abstract syntax trees), conveying only the static structure of the program. In contrast, the detection of run-time errors requires a representation that exposes run-time behavior of the program, such as its control flow, call-graph, etc. Such a fine-tuned representation enables the natural use of logic quantification and unification to reason over the program parts, mostly because logic unification establishes a pattern-matching scheme over the program's representation. However, many pattern detection problems can benefit from the availability of a structural as well as behavioral representation about the application. A general-purpose program investigation tool would thus need to provide different representations of the software application. However, such an approach will typically hamper the adoption by developers since they must have detailed knowledge about many different representations and explicitly quantify over them. The resulting queries are often overly complex.

To reconcile the need for different program representations with the ease of specification of queries, we have extended a logic program query language with a template-based pattern specification mechanism. This template-based meta-programming system allows the detection of patterns in software that require diverse representations of the code (i.e. structural as well as behavioral representations). Developers can specify templates that represent prototypical implementations of the patterns they wish to detect. The system then matches these templates on the static source code structure of the system as well as on a call-graph and a points-to analysis of the code. In particular, the call-graph and points-to analysis-based representations allow to conceal the matching of a sequence of statements and expressions into the matching on control-flow and run-time values respectively. As a result, the system allows the detection of structural code patterns as well as run-time errors in the program and provides a uniform pattern specification mechanism to express both. Furthermore, the templates can be embedded in logic queries, thereby still enabling developers to compose different patterns using logic operators and resort to pure query programs whenever the need arises. Most importantly, developers are not required to understand the intricate details of the structural and behavioral representations of the program that are required to perform the pattern detection.

The remainder of this paper demonstrates the technique of Soul's template-based queries by applying it to detect an example problem pattern in the code of a software system.

2. Undesired Object Mutability

The example we use in this paper is a particular run-time bug which we will refer to as *undesired object mutability*. An illustration of this bug is shown in Figure 1. It pertains to a *blackboard* architecture [2] that provides a simple data exchange mechanism between multiple components by means of a common infrastructure through which the components can exchange data. In our example, this infrastructure is implemented by a class `Blackboard` that offers a method `publish` to share a particular object, and a method `getLast` that returns the last published object.

One particular threat when using this simple architecture is that changes to a data object *after* it has been published can result in erratic behavior. For example, if a data object has been changed *in between* retrievals by different components, these components will share inconsistent data, compromising the application's correctness. In Figure 1, we illustrate the two possible occurrences of this bad smell: at the sending component and at the receiving component. The data object that can be exchanged is implemented by the class `ImmutableObject`. This class consists of two fields, namely `date` and `contents` along with their respective getter and setter methods. We have a component 1 that at some point in time creates a new `ImmutableObject` and publishes this object to the blackboard. A first occurrence of the undesired object mutability problem can be seen when, some time *after* the publish of the object happened, component 1 alters the state of the published object (in this case by changing the value of the `contents` field). Note that this call to the `setContents` method does not have to occur immediately after the object is published. Similarly, if in component 2 a data object is retrieved from the blackboard, this component 2 should not be allowed to alter the state of the object (this is the top-most violation indicated in component 2).

An important challenge in the detection of this bug is that it is not sufficient to only consider the direct state of `ImmutableObject`. This is illustrated by the bottom-most violation in component 2 of Figure 1. In this case, component 2 has created an internal reference to the value of the `date` field of the immutable object. The violation arises since the component alters the date object, which ultimately belongs to the state of the `ImmutableObject`. State changes must thus be prevented transitively.

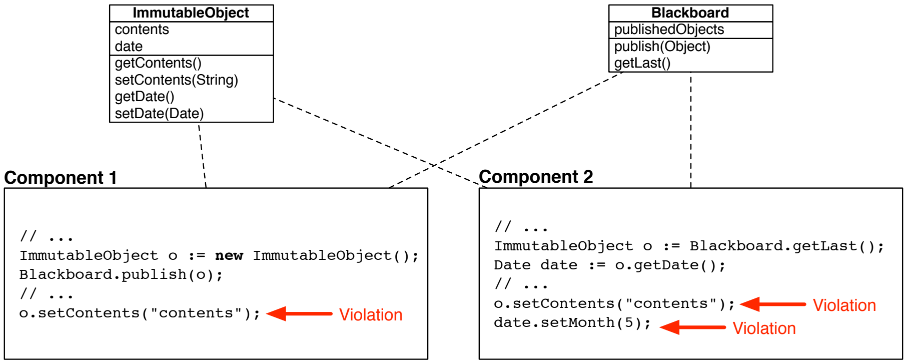
The following two Sections discuss the implementation of a query using both normal SOUL queries as well as template queries that detects occurrences of undesired object mutability.

3. Soul Queries

The "Smalltalk Open Unification Language" (SOUL) [14] is a logic program query language implemented in —and tightly integrated with— Smalltalk. SOUL programs are a hybrid combination of Prolog and Smalltalk, meaning that a SOUL program comprises Prolog conditions as well as Smalltalk expressions. This also entails that SOUL programs can manipulate any Smalltalk object as a logic value (i.e. as a constant term) and that these values can be exchanged transparently between logic conditions and Smalltalk expressions.

The SOUL programs and queries presented in this paper employ SOUL's symbiotic syntax [6], which closely resembles Smalltalk's keyworded message syntax. An expression such as `?a plus:?b is:?c` is a logic condition that imposes the predicate `plus:is:` over the logic variables *?a*, *?b* and *?c*. It states that the value of *?c* must be the sum of the values of *?a* and *?b*. Apart from this particular syntax, SOUL logic programs are evaluated exactly like Prolog programs. For example, we can use the aforementioned predicate in the query `if 2 plus:3 is:?result` to calculate the sum of 2 and 3. Evaluation of this query will produce the result 5, bound to variable *?result*.

Logic-based program queries rely upon a logic problem-solving strategy for their evaluation. One of the essential operations involved in this problem-solving strategy is *unification*. Unification is a pairwise matching process between logic terms that does not only establish an equivalence between logic values but also assigns values to variables. Two logic values are said to unify if they match –or– if an appropriate set of bindings can be found for the enclosed variables



```
// ...
ImmutableObject o := new ImmutableObject();
Blackboard.publish(o);
// ...
o.setContents("contents");
```

```
// ...
ImmutableObject o := Blackboard.getLast();
Date date := o.getDate();
// ...
o.setContents("contents");
date.setMonth(5);
```

Figure 1. Illustration of the Object Mutability pattern

```
?c classDeclarationHasName: {ImmutableObject},
?c hasTransitiveState: ?field,
?creation isExpression,
?creation equals: classInstanceCreation(?,?,?c,?,?),
?someClass definesMethod: ?method,
?method invokes: ?invocation calling: ?publishMethod,
?publishMethod methodDeclarationHasName: {publish},
?invocation methodInvocationHasArguments: <?argument>,
?argument mayAliasWith: ?creation,
?method invokesTransitive: ?mutatorInv after: ?invocation,
?mutatorInv methodInvocationHasExpression: ?receiver,
or(?receiver mayAliasWith: ?creation,
   ?receiver mayAliasWith: ?field),
?mutatorInv calls: ?mutatorMethod,
?mutatorMethod writesTo: ?field
```

Figure 2. A direct query to detect undesired object mutations

such that they match when each variable's binding substitutes for the same variable in both values. Using SOUL's standard definition of unification, which is identical to the one of Prolog and which uses object identity to match programs (i.e. parse tree entities), we can use the query in Figure 2 to detect the undesired object mutation bug.

Lines 1–2 gather all field declarations ?field in ImmutableObject, or transitively contained objects. The remainder of the query searches for the invocations and assignments to those fields, following an invocation of the publish method. To this extent, lines 3–9 gather all methods ?method that perform an invocation of the publish method using an instance of ImmutableObject as its argument. Lines 10–13 identify method invocations *after* the invocation of the publish method on receivers ?receiver that are values of the transitive state of the ImmutableObject instance. In addition, lines 14–15 ensure that the invoked method actually performs an assignment to such a transitively contained field declaration.

This query mostly reasons about the static structure of the program's source code, but requires quantification over call-graph, control-flow graph and points-to analysis. The latter is used by the red mayAliasWith: predicate that determines whether two expressions that are syntactically different can have coinciding runtime values. This points-to analysis is obtained through the Spark [11] toolkit of the Soot Java Optimization Framework which implements a conservative, flow-insensitive and context-insensitive points-to analysis. Similarly, the green predicates on lines 10 and 14 quantify over a control-flow and call-graph representation of the system.

Although all these representations of the system can be uniformly quantified over in SOUL, this explicit quantification renders the query of Figure 2 extremely verbose. In other words, the query does not convey the actual pattern it tries to match. To resolve this issue, we have — as described in detail in [1] — integrated the results of the points-to analysis in the unification procedure of SOUL itself. This reduces the complexity of the query in Figure 2 because we can omit the explicit quantification using the mayAlias: predicate. Furthermore, SOUL offers the ability to express the same query using concrete source code syntax patterns, as we will describe in the next section.

4. Template Queries

```
1  ?c classDeclarationHasName: {ImmutableObject},
2  ?c hasTransitiveState: ?field,
3  
4  jtExpression(?o){ new ?c(?)},
5  
6  jtMethodDeclaration(?m){
7    ?modList ?return ?name(?argList) {
8    ?exp.publish(?o);
9    ?some.?invocation(?aList); }},
10 
11 
12 jtClassDeclaration(?){
13  class ? {
14   ?mod2List ?return2 ?invocation(?arg2List) {
15     ?field = ?exp2;}
16  }}
```

Figure 3. Template query for detecting undesired object mutations

Template queries are an extension to the SOUL language that allows developers to write program queries by means of specifying a concrete source-code template — representing a prototypical implementation of the pattern to be queried — instead of a Prolog program. As such, developers are able to express their intent by writing down their query as a piece of source code, in which points of variation can be indicated using logic variables. As we already discussed in the section above, SOUL offers the possibility to, next to querying the structure of a program, also use information resulting from a call-graph analysis and a points-to analysis into the reasoning process. In contrast to the regular SOUL queries, in which this semantical information needed to be explicitly quantified over, template queries hide the users of the query language from the details of all different program representations. Instead of matching the template query purely structurally, the semantic analyses are taken into account in the matching process of the template query.

Figure 3 is a template query that searches for the undesired object mutation bug. It reports those methods ?m that possibly write to a (transitively contained) ?field of an instance ?o of ImmutableObject after an publish message has been sent. The query consists of five conditions. The first two conditions (lines 1–2) are identical to the query in the previous section. The three other conditions of the predicate make use of source-code templates. The first template (on line 4 of the example) matches all Java expressions ?o that create a new instance of class ?c (ImmutableObject).

The second template (lines 6–9) identifies all method declarations ?m that invoke a publish of the object ?o and that is followed by an invocation ?invocation. The reasoning process behind these source-code queries will make use of the call-graph information that is made available such that not only calls immediately after the invocation of the ?publish method are considered, but also calls that occur later in the call-graph after the ?publish. Moreover, since the information from the points-to analysis is taken into account, the argument ?o of the invocation of

?publish will not only unify with the direct result of the instantiation of the immutable object, but also with all values that — according to the points-to analysis — possibly alias with this instantiation.

Finally, the third template (lines 12–16) restricts the bindings of the ?invocation variable to those that correspond to an invocation of a setter method, by selecting all methods that assign to the field ?field. In line 2 of the query, this variable ?field is bound to all transitive fields of the class ?c, of which ?o is an instance. Note that the unification of the ?invocation variable does not merely happen by comparing the method name of the setter method, but that the call-graph information is used to verify that the ?invocation effectively is invoked from within the control flow of method ?m.

To summarize, this predicate will return the methods ?m that invoke a setter method ?invocation for a field ?field of a data object, *after* a value ?o, that aliases with a data object, has been published.

5. Acknowledgements

Johan Brichau is funded by a "FIRST" post-doc grant of the Région Wallonne, Belgium. Andy Kellens and Coen De Roover are funded by research grants provided by the by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen). This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy.

References

[1] J. Brichau, C. De Roover, and K. Mens. Open unification for program query languages. In H. Astudillo and E. Tanter, editors, *Proceedings of the 16th International Conference of the Chilean Computer Science Society*, pages 92–101. IEEE Computer Society, 2007. (acceptance rate 34%).

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[3] J. Cohen and T. J. Hickey. Parsing and compiling using prolog. *Transactions on Programming Languages and Systems*, 9(2):125–163, 1987.

[4] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proc. of the Conf. on Domain-Specific Languages*, pages 229–242, 1997.

[5] C. De Roover, I. Michiels, K. Gybels, K. Gybels, and T. D'Hondt. An approach to high-level behavioral program documentation allowing lightweight verification. In *Proc. of the 14th IEEE Int. Conf. on Program Comprehension*, pages 202–211, 2006.

[6] M. D'Hondt, K. Gybels, and V. Jonckers. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In *Proc. of the 2004 Symp. on Applied computing*, pages 1328–1335, 2004.

[7] J. Fabry and T. Mens. Language-independent detection of object-oriented design patterns. *Elsevier Int. Journal on Computer Languages, Systems & Structures*, 30(1-2):21–33, 2004.

[8] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In *Proc. of the 20th European Conf. on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, 2006.

[9] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *Proc. of the 2nd Int. Conf. on Aspect-oriented software development*, pages 178–187, 2003.

[10] C. Lewerentz and F. Simon. A Product Metrics Tool Integrated into a Software Development Environment. In *Proc. of the ECOOP Workshop on Object-Oriented Technology*, volume 1543, pages 256–257, 1998.

[11] O. Lhoták. Spark: A flexible points-to analysis framework for java. Master's thesis, McGill University, December 2002.

[12] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proc. of the Conf. on Object-oriented Programming Systems, Languages and Applications*, pages 365–383, 2005.

[13] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Proc. of the 13th Int. Software Engineering and Knowledge Engineering Conf.*, 2001.

[14] SOUL:. http://prog.vub.ac.be/SOUL/.

[15] T. Tourwé and T. Mens. Identifying refactoring opportunities using logic meta programming. In *Proc. of the 7th European Conf. on Software Maintenance and Reengineering*, pages 91–100, 2003.

[16] 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.