Combining Fuzzy Logic and Behavioral Similarity for Non-Strict Program Validation

Coen De Roover [*]
Programming Technology Lab
Vrije Universiteit Brussel
cderoove@vub.ac.be

Johan Brichau
Laboratoire d'Informatique Fondamentale
Université de Lille
johan.brichau@lifl.fr

Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel
tjdhondt@vub.ac.be

Abstract

The quality of an application's implementation can be assured by validating the presence or absence of a set of user-prescribed software patterns such as software engineering best practices, programming conventions and indications of poor programming. Most of the existing pattern detection techniques, however, interpret pattern descriptions in an inflexible manner, leaving the quality assurance tool to approve only the most strictly adhering pattern implementations. In order to detect various concrete pattern implementations using a single pattern description, we have combined logic meta programming —wherein patterns can be expressed as constraints over facts representing a program's source code—, fuzzy logic and static program analysis in a way that is completely transparent to the end-user. We have achieved this by having the conditions in a logic rule interpreted as constraints over the run-time behavior source code constructs give rise to instead of as constraints over the literal source code constructs themselves. This way, a pattern's abstract description often suffices to recognize various concrete implementation variants with an indication of the similarity between the recognized implementation and the abstract pattern description.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis; D.1 [*Programming Techniques*]: Logic Programming

General Terms Design, Languages, Algorithms, Verification

Keywords Fuzzy Logic Programming, Logic Meta Programming, Program Validation, Quality Assurance of Object-Oriented Programs, Points-to Analysis, Program Analysis

1. Introduction

The quality of an application's implementation can be assured by validating the presence or absence of user-prescribed software patterns in its source code. Such patterns typically express either software engineering best practices [3], the programming style and conventions developers have agreed upon, or indications of bad design [7] suggesting the need for a thorough refactoring of an application's internals.

Software patterns are in essence abstractions of often recurring concrete sequences of programming constructs. As is common to all abstractions, details are omitted from pattern descriptions until only the essence of the software pattern is left. In practice, concrete pattern implementations often deviate from the prototypical implementation of the abstracted pattern description. These deviations can range from minor differences in naming conventions, over differences in the employed datastructures, to the combination of multiple patterns in one implementation unit.

Most of the existing pattern detection tools interpret machine-readable transcriptions of abstracted pattern descriptions however in a strict all-or-nothing manner [1], leaving them to detect only the most literal pattern implementations successfully. There is little room for partial adherence of concrete source code to a given abstract pattern description as the employed techniques are unable to overcome any discrepancies between the evidence necessary to prove the presence of a pattern and the evidence from the source code at hand.

To work around the limited support of existing pattern detection techniques for dealing with these discrepancies, developers typically have to resort to describing every possible concrete implementation variant of an abstract pattern description. This is not only a far from elegant, but also an inadequate solution to the problem: in general, it is impossible to predict every conceivable implementation variant of a pattern while transcribing its abstract description to a format understood by the pattern detection tool.

We therefore desire the machine-readable transcription of a software pattern's description to be:

- As close as possible to the abstract description of the pattern and as far as possible from concrete source code implementation variants.
- Interpreted by the pattern detection tool in a non-strict manner such that the abstract description of the pattern suffices to detect most concrete implementation variants with an indication of the degree of adherence of the implementation to the pattern description.

1.1 Outline of the paper

We will start our discussion in Section 2.1 with a concrete example of the aforementioned problem. Subsequently, we will present a high-level overview of our solution to this problem in Section 2.2. Section 3 will detail the implementation of the fuzzy logic meta

* Ph.D. scholarship funded by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

[1] Notable exceptions will be reviewed in the Related Work section.

```
class Y {
  private X var;
  public X getVar { return var; }
  public void setVar(X val) { var = val; }
}
```

Figure 1. Prototypical getter and setter method best practice pattern implementations in Java.

programming platform supporting our approach. Its subsections detail and motivate each cornerstone of the platform. Section 3.1, for instance, describes the typical logic meta programming setup. Section 3.2 outlines the theory and practice behind fuzzy logic programming, while Section 3.3 describes how the platform incorporates information about a program's possible run-time behavior into its reasoning process. Section 4 will evaluate the effectiveness of the resulting fuzzy logic meta programming platform in checking a program's adherence to some well-known object-oriented best practice patterns. Related work will be reviewed in Section 5 and we will conclude with a discussion of our approach in Section 6.

2. Problem Discussion

Before presenting our solution to the pattern detection problem identified in the introduction, we will illustrate its importance for quality assurance by means of a concrete motivating example.

2.1 A Motivating Example

Consider detecting violations of the *"no direct instance variable accesses outside getter and setter methods"* principle as an example motivating the need for a more flexible detection of software patterns for quality assurance. The getter and setter method best practice patterns [3] advocate indirect access to instance variables through calls to getter and setter methods which respectively return and set the value of the variable they are protecting. The purpose of these methods is to ease evolving classes as they allow the internals of a class to be changed without having to modify all uses of this class throughout the application.

Methods `getVar()` and `setVar(X)`, shown in Figure 1, correspond to the prototypical Java implementations of the getter and setter method best practice patterns for the instance variable `var`.

Given a machine-readable transcription of the getter and setter method patterns' descriptions, one can employ a pattern detection tool to enforce the consistent use of these methods throughout an application. In a quality assurance setting, one can for instance require that the only methods allowed to access an instance variable directly, are methods that were recognized by the pattern detection tool as implementing the getter or setter method best practice pattern. Methods not adhering to this best practice can be considered indications of a suboptimal implementation which renders their identification worthwhile to ensure an application's quality.

Most pattern detection tools interpret machine-readable transcriptions of a pattern's description in a strict all-or-nothing manner, leaving them to detect only the implementations that strictly adhere to the pattern's prototypical implementation. However, in practice, pattern implementations often deviate from this prototypical implementation. Consider for instance the `Person` class depicted in Figure 2. While the `getAge()` and `setBirthday(newDay)` methods semantically adhere to the description of the getter and setter method best practice patterns, their source code deviates from the prototypical implementation. The `age` instance variable depends on the `birthday` instance variable and thus has to be recalculated when the latter changes, but this recalculation is only performed —out of performance considerations— when a client

```
class Person {
  private Date birthday;
  private int age;
  private boolean ageDirty;
  public Date getBirthday() {
    return birthday;
  }
  public void setBirthday(Date newDay) {
    ageDirty = false; birthday = newDay;
  }
  public int getAge() {
    if(ageDirty) age = ...;
    return age;
  }
}
```

Figure 2. Deviating getter and setter method best practice pattern implementations in Java.

accesses the person's age and this calculation hasn't been performed yet.

Although the aforementioned methods semantically adhere to the getter and setter method best practice pattern descriptions, they can not be detected as such by detection techniques relying on their prototypical implementation. A quality assurance tool would thus wrongly flag them as violating the *"no direct instance variable accesses outside getter and setter methods"* principle. We would therefore like the employed pattern detection technique to recognize multiple concrete implementations deviating from the prototypical pattern implementation.

2.2 Overview of our solution

We employ a logic programming language as the meta language to reason about object-oriented programs. In such a logic meta programming approach, descriptions of software patterns are expressed in a machine-readable format as logic rules over logic facts representing source code constructs. Detecting the presence of a pattern in the source code therefore amounts to finding all solutions to a logic query.

A logic meta programming approach is however not exempt from the problems common to most pattern detection techniques. The problems identified in the previous section manifest themselves in the following ways:

- Logic rules express constraints over concrete source code constructs and are thus by nature closer to one of the prototypical implementations of a software pattern than to its general abstract description. Multiple rules are therefore needed to detect different implementation variants of the same pattern.
- The resolution procedure used to find solutions to a logic query can only end with either complete success or failure. It is therefore inadequate to find instances of a software pattern which deviate from the prototypical implementation described by a logic rule.

To overcome these problems, our solution comprises two complementary extensions to the typical logic meta programming setup: we extend the resolution procedure to handle partial truths and we extend the unification procedure to take information about a program's possible run-time behavior into account.

Fuzzy Logic Programming

The first cornerstone of our solution to the above problems comprises the use of a fuzzy logic programming language which allows vague concepts and partial truths to be modeled explicitly and which is able to draw sensible conclusions from premises that are only partially true.

A set's characteristic function, indicating whether an element belongs to a set or not, is generalised to support gradual set membership in fuzzy set theory [27]. Translated to logic meta programming, this concept allows facts and rules to be annotated with partial truth degrees. If they wish to do so, users can still write a separate logic rule for each pattern implementation variant, but will now also be able to express their confidence in each rule by a truth degree representative for the amount of false positives that is to be expected from the heuristics employed in its body.

In addition, a fuzzified resolution procedure is able to draw conclusions with a varying degree of truth from rules whose body is only partially satisfied. The rules themselves, however, still express constraints over source code elements which renders the rules by nature closer to a concrete pattern implementation variant than to the general abstracted pattern description.

Information about a Program's Run-time Behavior

The second cornerstone of our solution to the above problems comprises incorporating information about a program's run-time behavior into the reasoning process. We do this in an end-user transparent way by extending the unification procedure to succeed even on syntactically different source code expressions whenever they might evaluate to overlapping sets of objects at run-time. As the object-oriented programming paradigm centers around objects communicating through messages, knowledge about the objects an expression evaluates to is crucial behavioral program information. However, without executing the program, such information can only approximate a program's actual run-time behavior. We will therefore introduce the notion of a fuzzy unification procedure. In the remainder of this section, we will discuss the motivation behind this choice in more detail.

As logic rules express constraints over a program's source code, they can easily become closely connected to a concrete implementation variant of the software pattern. In order to detect multiple pattern implementation variants using only one logic rule, we would rather have this rule referring to information about the pattern's run-time behavior instead of one of the concrete source code implementation variants giving rise to this behavior.

While information about a program's run-time behavior can be obtained through a dynamic analysis, it requires the program's execution to be monitored rendering the obtained information valid for only one out of many possible program executions. We therefore prefer to obtain information about a program's run-time behavior through a static analysis. The thus obtained information is valid for all possible program executions, but might only be approximating the application's actual behavior in order to achieve this generality.

As static analyses have been traditionally applied in program optimisation and verification settings, many developers are unacquainted with the way these analyses represent behavioral information and don't always know how to interpret the analysis results correctly. We would rather incorporate this information in a way that is completely transparent to the user. Conditions in a user's logic rule can therefore be interpreted as constraints over the run-time behavior source code constructs give rise to instead of as constraints over the literal source code constructs themselves. This way, users can keep on expressing patterns as logic rules the way they were used to, but will now also be able to detect other pattern implementation variants giving rise to the same run-time behavior.

We achieve this more semantic interpretation of logic rules by extending the unification procedure to take behavioral information into account. The extended unification procedure will unify two syntactically different source code expressions whenever they might evaluate to overlapping sets of objects at run-time.

Since we are obtaining behavioral information through a static analysis which can only approximate a program's actual run-time behavior, the extended unification procedure can however not always succeed with a total truth degree. Whenever source code expressions are syntactically equivalent, their unification will succeed completely as usual. If the static analysis driving the extended unification procedure has however determined that two syntactically different expressions *may* point to overlapping sets of objects at run-time, the expressions can be unified only with a partial degree of truth. If the static analysis however derived that two expressions can never alias, the unification procedure fails. We therefore need the approximate reasoning capabilities offered by a fuzzy logic programming language.

3. Fuzzy Logic Meta Programming

In this section, we will detail the implementation of our fuzzy logic meta programming platform. First of all, we will describe how logic programming can be employed to reason about object-oriented programs. We will continue with a short introduction to fuzzy logic and describe how we applied this theory to alter the platform's resolution procedure in order to handle partial truths. Finally, we will outline how we altered the platform's unification procedure to take information about a program's possible behavior into account.

3.1 Logic Meta Programming Setup

We employ a logic programming language as the meta language to reason about object-oriented programs. Over the years, this approach has been applied to a variety of problems in object-oriented software engineering. Some examples include: reasoning about object-oriented design [26], checking and enforcing programming patterns [19], supporting the evolution of software applications [20] and identifying software refactoring opportunities [25]. While these researchers have employed the SOUL logic meta programming system to reason about Smalltalk programs, we will use a variant [6] of the system for the Java programming language in this paper.

The advantages of a logic programming language for meta programming purposes are already well-known [26]. First of all, predicates in logic programming languages describe relations between their arguments in a declarative instead of an operational manner. Logic programming languages are, furthermore, extremely suited to reasoning about source code thanks to their advanced pattern matching abilities, built-in backtracking and support for powerful programming concepts such as recursion.

To allow the use of logic queries to reason about an object-oriented program, a logic meta programming approach represents the base program as logic facts according to a meta model. In a structural meta model, these facts state the classes and methods present in the program. Figure 3 depicts most of the functors we will encounter in the logic representation of the statements and expressions in a method's body, while Figure 4 shows an example of such a reified method body. Every Java expression in the parse tree is typed, in contrast to statements which do not return a value.

The nodes of a method's parse tree are reified as special logic terms for which unification with ordinary logic functors still succeeds. However, they contain additional information, such as the parent of the node they are representing or its defining method. As is demonstrated in Figure 5, this enables us to query the node for its parent in the parse tree without having to perform an explicit parse tree walk. The `?s` variable[2] is for instance bound to a parse tree

[2] The syntax of the SOUL logic meta programming platform is slightly different from the prototypical Prolog one. Logic variables start with a question mark, standalone question marks denote anonymous variables, while lists are denoted as: `<1,?number,?,2>`.

```
send(?type, ?receiver, ?message ?argumentList)
assign(?type, ?operator, ?lhs, ?rhs)
variable(?type,?identifier)
literal(?type,?value)
binaryExp(?type, ?operator, ?lOperand, ?rOperand)
new(?type, ?class, ?argumentList)
return(?expression)
ifte(?condition, ?trueBlock, ?falseBlock)
for(?init, ?expression, ?update, ?body)
...
```

Figure 3. Extract of the logic meta model according to which method parse trees are reified.

```
class Bar {
    float x;
    public void foo() {
      x = 42 + 3.14d;
    }
}

methodStatements(?m,?s),
?s = statements(<assign('float', '=',
                  variable('float', 'x'),
                  binaryExp('double','+',
                     literal('int', '42'),
                     literal('double', '3.14d')))>)
```

Figure 4. A method's reified body.

```
class Foo {
  public Integer getSum() {
    return sum;
  }
}

if statementInMethod(?s, ?),
   ?s = return(variable(?type, 'sum')),
   surroundingMethodName(?s, ?methodName).

Solutions: ?methodName -> 'getSum'
           ?type -> 'java.lang.Integer'
```

Figure 5. A Java method, a logic query, and the corresponding solutions.

node logic term and unifies with a `return(variable(?type, 'sum'))` functor, while we can still query the return statement for the name of its surrounding method.

Starting from the basic structural facts, more complex relationships can be derived by defining the appropriate logic rules. The following rules, for instance, express what it means for one class to be an (in)direct subclass of another:

```
isInHierarchyOf(?directSubclass, ?root) if
  isSubClassOf(?directSubclass, ?root).

isInHierarchyOf(?indirectSubclass, ?root) if
  isSubClassOf(?indirectSubclass, ?parent),
  isInHierarchyOf(?parent, ?root)
```

The first rule expresses that one class is in the class hierarchy of another class when it is the subclass of that class. The second rule expresses that subclasses of a class which is in the hierarchy of some class are also in the hierarchy of that same class.

One can use the `isInHierarchyOf/2` predicate in logic queries both to *verify* whether there is a hierarchy relationship between two classes and to *detect* the classes another class has in its hierarchy. By binding both variables in the `isInHierarchyOf/2` predicate, we can for instance verify whether the `testapp.SumComponent-Visitor` class is a subclass of `java.lang.Object`. By only binding the `?root` variable, we can find all subclasses of the `java.lang.Object` class.

As we will demonstrate in Section 4, the multi-directional nature of logic programming languages is especially useful for quality assurance where logic rules describe user-prescibed software patterns a program should adhere to. A single rule can be used to verify whether source code adheres to the constraints imposed by a pattern, to find source code adhering to or violating the pattern, to find all occurrences of patterns in a particular piece of the source code, or to find all occurrences of patterns in the entire application.

3.2 Fuzzy Logic Programming

Should the logic meta programming setup described above employ a strict all-or-nothing resolution procedure, it would be unable to find pattern implementation variants that do not completely adhere to all constraints over parse tree nodes expressed in a rule's body. As we already explained in Section 2.2, our meta programming platform therefore employs a fuzzy logic programming language which allows partial truths to be modeled explicitly and can draw sensible conclusions from rules whose bodies are only partially satisfied.

3.2.1 Fuzzy Logic

Analogous to the partial set membership degrees which can be assigned to the elements of a fuzzy set [27], fuzzy logics [11] assign a degree of truth to logic propositions. One proposition may be absolutely true, while another may evaluate to a truth degree between absolute truth and absolute falsity. In sharp contrast to probabilistic logics, fuzzy logics are truth-functional: the truth of a formula is determined by the truth of its constituents. As different semantics can however be given to the logical connectives $\wedge$, $\vee$ and $\neg$, there exist many different kinds of fuzzy logic. As a notion of proof, these logics use modus ponens.

In fuzzy logic programming, a resolution procedure is used as the notion of proof. Lee [16] was the first to extend the classical resolution procedure to handle partial truths, initiating a plethora of "Fuzzy Prolog" systems. These all differ in the way they model the logical connectives as well as in whether or not they allow fuzzy facts, fuzzy rules or fuzzy constants. A detailed historical overview of the resulting programming languages can be found in Alsinet's dissertation [1].

3.2.2 Syntax and Semantics

Our logic meta programming platform employs a fuzzy logic programming language which allows facts and rules to be annotated by partial truth values. It is close to the *f-Prolog* system [18] in that it uses a similar fuzzy resolution procedure. Our current implementation however only supports real-valued partial truths in the interval 0, 1, while the *f-Prolog* system in addition supports fuzzy numbers. A detailed overview of our implementation can be found in [5].

The fuzzy resolution procedure only differs from the crisp resolution procedure in the quantification of the deduced answer sets for a goal. We can therefore summarize it briefly by describing how the truth degree associated with an answer set is calculated. Weighted logic rules are of the form:

$$q \quad c \text{ if } q_1, \ldots, q_n. \text{ where } c \in 0, 1$$

The truth degree of the conclusion q can be computed as the product of c and the minimum of the truth degrees of the subgoals $q_1 \ldots q_n$. Conjunction, disjunction and implication are thus modeled as minimum, maximum and product respectively. We will interpret c as the confidence we have in the conclusion q given the absolute truth of its subgoals $q_1 \ldots q_n$.

3.2.3 Fuzzy Logic Programming in Practice

Consider, as an introductory example, the fuzzy logic program shown below. It models the vague concept of a grocery item's popularity: any product of which more than 10 items have been sold must definitely be a popular product, while other products will most likely become popular given an attractive packaging and good advertising. Its background information states that 15 `flowers` have already been sold, while the product `chips` has a fairly attractive packaging and has been reasonably well advertised.

```
sold(flowers, 15).
attractive_packaging(chips) : 0.9.
well_advertised(chips) : 0.6.

popular_product(?product) if
  sold(?product, ?amount),
  ?amount > 10.

popular_product(?product) : 0.8 if
  attractive_packaging(?product),
  well_advertised(?product).
```

For this program, we can derive that the product `chips` must be fairly popular, as we find it as a solution to the query `if popular_product(?product)` with a reasonably large partial truth degree of $\min(0.9, 0.6) \cdot 0.8 = 0.48$. We can, on the other hand, be absolutely certain about `flowers` being a popular product.

In our experiments, we found that it is sometimes useful to explicitly assign a truth value to the body of a rule. It could for instance be argued that the average of the truth degrees of the subgoals `attractive_packaging` and `well_advertised` is a more balanced measure for the popularity of a product than the default semantics of minimum for conjunction. We can express this as follows:

```
popular_product(?product) : 0.9 if
  nicely_packed(?product) : ?c1,
  well_advertised(?product) : ?c2,
  (?c1 + ?c2) / 2.
```

The truth degree of each subgoal is 1 except for the last one, which will be the average of the partial truth degrees of `attractive_packaging` and `well_advertised`. Using this rule, we obtain a truth of $0.9 \cdot \min(1, 1, \frac{0.9+0.6}{2}) = 0.675$ for the popularity of the `chips` product.

3.3 Incorporating Information about Possible Run-time Behavior

In order to detect multiple pattern implementation variants using a single logic rule, we let our fuzzy logic meta programming platform interpret conditions in a rule as constraints over the run-time behavior source code constructs give rise to instead of as constraints over the literal source code constructs themselves. We implemented this more semantic interpretation of logic rules by extending the unification procedure to take behavioral information into account, which is the topic of this section.

3.3.1 Information about Run-time Behavior

As the behavior of an object-oriented program is governed by the interactions between the run-time entities it is composed of, knowledge about the objects a reference might point to is crucial to any form of reasoning about the behavior of the program. This information can be derived through a points-to analysis [23] on the object-oriented program's source code. Such a static analysis computes at compile-time the set of all heap objects a reference might point to at run-time during an execution of the program.

We obtain this vital kind of information through the Spark [17] toolkit of the Soot Java Optimization Framework. It implements a conservative flow-insensitive, context-insensitive points-to analysis for Java. Informally, a flow-insensitive analysis disregards the order of statements in a method and is thus unable to take strong updates to variables into account (e.g. an assignment overriding a previous variable assignment). A context-insensitive analysis on the other hand, doesn't perform a separate analysis for the different calling contexts of a method. Although flow-insensitiveness and context-insensitiveness are one of the major sources of imprecision in the analysis of object-oriented programs, they also guarantee a reasonably efficient computation [12] —while the precision of the analysis results suffices for our purposes. Indeed, we will explicitly embrace the idea that we are only handling information about a program's *possible* run-time behavior by attaching a lower degree of truth to solutions of logic queries which were found solely thanks to this information.

3.3.2 Similarity-Based Unification

In Section 3.2.2, we described a generalisation of the crisp resolution procedure that is able to draw sensible conclusions from rules whose body is only partially satisfied. An analogous extension of the crisp unification algorithm allows for two incompatible logic terms to be unified in case they can be considered semantically or syntactically similar up to a certain degree. The similarity degree associated with the most general unifier calculated by this algorithm can be used further on in a "Fuzzy Prolog" system supporting partial truths.

Sessa's weak unification algorithm [24], for instance, relies on a user-provided similarity relation between function and predicate symbols to overcome syntactical failures of the refutation process in case the symbols under investigation are deemed similar up to a certain degree.

3.3.3 Behavioral Similarity for Fuzzy Logic Meta Programming

Our fuzzy logic meta programming platform utilizes an extended variant of the weak unification algorithm described above. Its similarity-based unification procedure will unify two syntactically different source code expressions in case they might evaluate to overlapping sets of objects at run-time.

Whenever two parse tree node logic terms need to be unified, we query the points-to analysis results for their respective points-to sets. Our unification procedure succeeds when these points-to sets have a non-empty intersection. Hence, we use a may-alias relation to gauge the similarity between parse tree node logic terms [3].

In order to identify the references in the analysis results corresponding to each parse tree node logic term unambiguously, we need some additional contextual information. As we explained in

[3] Our prototype implementation currently only computes aliasing information between `this` references; the value returned by return statements; references to field, static, local variables and arguments; array references; and receivers and results of message sends. While this already suffices for the use cases described in Section 4, we need to generalise the computation to other parse tree nodes in future work.

Section 3.1, these terms can be queried for additional information about the parse tree node they represent without having to perform an explicit parse tree walk. Among others, we use this information to get a node's defining method and class.

Since we are obtaining behavioral information through a points-to analysis which approximates a program's actual run-time behavior conservatively, the extended unification procedure can however not always succeed with a total unification degree. Whenever parse tree node logic terms are syntactically equivalent, unification will succeed on their functor-representation just as it would under the classical crisp unification algorithm. In case the classical unification algorithm failed, but if we were able to determine through the points-to analysis that the parse tree nodes might actually evaluate to overlapping sets of objects at run-time, we let their unification succeed with a partial unification degree of 0.5 without unifying any of the variables in their functor representations. If the analysis however determined that this can never be the case, the unification procedure fails after all. As partial unification degrees are propagated by the fuzzy resolution procedure, they will influence the truth degrees associated with each of a query's solutions.

As a result, our fuzzy logic meta programming platform interprets the conditions in a rule as constraints over the run-time behavior source code constructs give rise to instead of as constraints over the literal source code constructs themselves. As an illustration, consider the following logic query:

```
if methodInClass(?method, ?class),
   statementInMethod(?statement, ?method),
   instanceVariableInClass(?instvar, ?class),
   statementReturns(?statement, ?instvar).
```

This query will identify all methods `?method` in a class `?class` that return an instance variable `?instvar` from that class. Its last condition demands that a return statement in the method's parse tree has the instance variable as its argument. Upon evaluation of the query, the `?instvar` logic variable —bound to a Java instance variable parse tree node logic term—, will be unified against the parse tree node corresponding to the argument expression of the return statement. Whenever both parse tree node logic terms are syntactically equivalent, the plain unification algorithm will succeed on their functor-representation. The query's last condition is thus interpreted as a constraint on the method's actual parse tree nodes. Under similarity-based unification, the unification algorithm will succeed also when there is a syntactic difference between the parse tree nodes as long as these parse tree nodes might actually evaluate to overlapping sets of objects at run-time. The return statement's argument expression is, in other words, allowed to be any complicated expression as long as it possibly evaluates at run-time to the instance variable. The query's last condition will thus be interpreted as a constraint on the possible values returned by the return statement.

As the unification procedure is intrinsic to the refutation process, users can continue expressing their software patterns as logic rules the way they were used. Their rules are, in fact, merely interpreted in a more flexible manner by taking a program's possible run-time behavior into account. Through this end-user transparent combination of logic meta programming, fuzzy logic and a heavy-weight program analysis, our platform is able to detect multiple concrete pattern implementations using a single abstract pattern description. We will demonstrate this property through the use cases in the next section.

4. Expressing Software Patterns as Logic Rules

Our fuzzy logic meta programming platform can be applied to problems in most of the domains logic meta programming has been applied to before. In this paper, we will however only evaluate the effectiveness of our prototype in a quality assurance setting where its ability to approve source code that deviates from the prototypical implementation of a software pattern with an indication of its degree of adherence is especially well appreciated.

```
getterMethod(?class, ?method, ?instvar) if
  methodInClass(?method, ?class),
  instanceVariableInClassChain(?instvar, ?class),
  variableName(?instvar, ?vname),
  methodStatements(?method, ?s),
  ?s = <return(variable(?vtype, ?vname))>).

getterMethod(?class, ?method, ?instvar) : 0.9 if
  methodInClass(?method, ?class),
  instanceVariableInClassChain(?instvar, ?class),
  statementInMethod(?statement, ?method),
  statementReturns(?statement, ?instvar).
```

Figure 6. Logic rules for getter methods.

Our prototype implementation comes with a library of predefined logic rules corresponding to well-known object-oriented software patterns [3, 7, 8], but can be extended by a user's own definitions in a straightforward manner. While assessing an application's adherence to this set of rules, multiple source code units adhering to a rule's conditions will be found with a varying degree of truth. Users can initially choose to consider only solutions that are absolutely true, ignoring the platform's ability to interpret the conditions in their logic rules more liberally. By lowering the truth degree threshold above which solutions are reported, users will also find implementations that meet a rule's conditions only partially.

The use cases described in this section primarily consist of best practice software patterns [3] including the getter method, setter method and double dispatching best practice patterns. For each pattern, we will demonstrate how its abstract description and prototypical implementation can be captured conveniently by a logic rule. In addition, we will demonstrate our platform's ability to detect different pattern implementations giving rise to similar run-time behavior by detecting these pattern instances in a Java application of which source code extracts are depicted in Figures 11, 12 and 13.

4.1 Getter Methods

Our first use case consists of detecting and enforcing a consistent use of the Getter Method [3] best practice pattern. As we have seen in Section 2.1, it advocates indirect access to instance variables through calls to getter methods which simply return the value of the variable they are protecting. The Java naming convention for these methods is to prefix the capitalized variable name with "get".

4.1.1 Logic Rule for Prototypical Implementation

The logic rules shown in Figure 6 express what it means for a method `?method` to be a getter method for an instance variable `?instvar` in class `?class`. As mentioned in Section 3.1, the multi-directional nature of the `getterMethod/3` logic predicate allows it to be used in queries verifying whether a method is a getter method as well as in queries finding all getter methods in the application, a specific class or for a specific instance variable.

The topmost rule in Figure 6 corresponds to the prototypical implementation of a getter method for an instance variable in a class:

```
class Y {
  private X var;
  public X getVar { return var; }
}
```

`if getterMethod(?class, ?method, ?var) : ?c`

?class	?var	?method	?c
SumCmpntVisitor	sum	getSum()	1
SumCmpntVisitor	sum	getSum()	0.9
SumCmpntVisitor	sum	getSum()	0.45
SumCmpntVisitor	sum	returnSum()	0.45
SumCmpntVisitor	sum	retrieveSum()	0.45
SumCmpntVisitor	sum	retrieveSum()	0.45

Table 1. Detected getter method instances.

The first two lines of the rule state that a getter method `?method` needs to be part of a class `?class` in whose hierarchy `?instvar` is an instance variable. In addition, the method's parse tree is required to unify exactly with the parse tree of the prototypical getter method implementation shown above. The `getSum()` method in Figure 11 is an example of a getter method that can be successfully detected using this rule since it adheres faithfully to the prototypical getter method implementation.

We can use the `getterMethod/3` predicate in queries to enforce the consistent use of getter methods throughout an application by requiring that the only methods allowed to access an instance variable directly, are methods that were recognized by the platform as a getter method. However, imagine a getter method that logs a message to a file before returning the instance variable it is protecting. Such a method cannot be detected as a getter method by the above logic rule and would thus be flagged by the quality assurance tool as violating the the *"no direct instance variable accesses outside getter methods"* principle.

4.1.2 Fuzzy Logic Rule for Alternative Implementations

To increase the likelihood of a method being recognized as an alternative getter method implementation, we can resort to a heuristic such as the one expressed in the rule near the bottom of Figure 6. This rule does not require a method's parse tree to match the parse tree of the prototypical getter method's implementation, but merely requires that the method contains a return statement with the instance variable as its argument. To indicate that getter method implementations detected by this rule do not follow the prototypical implementation, we have annotated the rule with a truth degree of 0.9 which expresses our trust in the heuristic it employs.

The `SumCmpntVisitor` class depicted in Figure 11 does, however, contain three –somewhat convoluted– alternative implementations of the getter method best practice pattern which cannot be detected when the conditions in the above rule's body are interpreted in a strict syntactical manner. The `returnSum()` method, for instance, can be classified semantically as a getter method as it indirectly returns the `sum` instance variable by invoking a recursive method which returns its first argument after the amount of recursive calls indicated by its second argument has been performed.

Thanks to the similarity-based unification described in Section 3.3, our fuzzy logic meta programming platform is however able to interpret the conditions in a rule as constraints over the runtime behavior source code constructs give rise to instead of as constraints over the literal source code constructs themselves. The rule's final condition will thus be interpreted as a constraint on the possible values returned by the return statement. The unification of the `?instvar` instance variable parse tree node logic term and the parse tree node corresponding to the argument expression of the return statement will succeed with a unification degree of 0.5 if the points-to analysis has determined that both nodes might alias at run-time.

Table 1 contains an overview of the getter method implementation variants recognized by our fuzzy logic meta programming platform. The `getSum()` method is detected once with a total truth degree originating from the topmost logic rule in Figure 6. It is also detected once by the logic rule near the bottom with a truth degree of 0.9 using plain unification. Finally, it is detected a third time with a truth degree of $0.45 = 0.9 \cdot \min(1, 1, 1, 0.5)$, originating from the second logic rule in combination with similarity-based unification. The `retrieveSum()` method is recognized twice with a truth degree of 0.45 due to the detected possible aliasing between `sum` and `value` on line 29 and the detected possible aliasing between `sum` and the `getSum()` call on line 30.

```
correctlyNamedGetterMethod(?c, ?m, ?var) if
  getterMethod(?c, ?m, ?var) : ?c1,
  selectorOfMethod(?selector, ?m),
  instanceVariableName(?var, ?varname),
  capitalized(?varname, ?cvarname),
  concat('get', ?cvarname, ?sel),
  similar(?sel, ?selector) : ?c2,
  ?c1 * ?c2.
```

Figure 7. Logic rule for named getter methods.

`if correctlyNamedGetterMethod(?c, ?m, ?var) : ?t`

?c	?var	?m	?t
SumCmpntVisitor	sum	getSum()	1
SumCmpntVisitor	sum	getSum()	0.9
SumCmpntVisitor	sum	getSum()	0.45
SumCmpntVisitor	sum	returnSum()	0.25
SumCmpntVisitor	sum	retrieveSum()	0.20
SumCmpntVisitor	sum	retrieveSum()	0.20

Table 2. Correctly named getter method instances.

4.1.3 Fuzzy Logic Rule for Naming Convention

As we mentioned in the beginning of this section, the Java naming convention for getter methods is the capitalized name of the protected instance variable prefixed by "get". The logic rule for the `correctlyNamedGetterMethod(?c, ?m, ?var)` predicate, shown in Figure 7, will find getter methods in an application's source code with an associated truth degree that is representative for the degree to which each pattern instance is well-named.

It relies on the fuzzy predicate `similar(?s1,?s2)` which succeeds with a truth value of:

$$1 - \frac{e(?s1, ?s2)}{max(|?s1|, |?s2|)}$$

where e is the Levensthein edit distance which, informally, calculates for two strings the amount of add, replace or delete operations necessary to transform one string into the other. Table 2 lists the degree to which each of the getter methods that were previously found is named well.

4.2 Setter Methods

Our second use case consists of detecting and enforcing the consistent use of the Setter Method [3] best practice pattern. Analogous to the Getter Method best practice pattern, it advocates indirect access to instance variables through calls to setter methods which simply assign their argument to the instance variable they are protecting.

4.2.1 Logic Rule for Prototypical Implementation

The logic rules shown in Figure 8 express what it means for a method `?method` to be a setter method for an instance variable `?instvar` in class `?class`. The topmost rule corresponds to the prototypical implementation of a setter method:

```
setterMethod(?class, ?method, ?instvar) if
  methodInClass(?method, ?class),
  argumentOfMethod(?argument, ?method),
  instanceVariableInClassChain(?instvar, ?class),
  variableName(?instvar, ?ivarname),
  variableName(?argument, ?argname),
  methodStatements(?method, ?s)
  ?s = <assign(?atype, ?aoperator,
               variable(?lhstype, ?ivarname),
               variable(?rhstype, ?argname))>).

setterMethod(?class, ?method, ?instvar, ?c) : 0.9 if
  methodInClass(?method, ?class),
  instanceVariableInClassChain(?instvar, ?class),
  argumentOfMethod(?argument, ?method),
  expressionInMethod(?expression, ?method),
  isAssignment(?expression, ?instvar, ?argument).
```

Figure 8. Logic rules for setter methods.

```
class Y {
  private X var;
  public void setVar(X val) { var = val; }
}
```

The first three lines of the rule state that a setter method `?method` has at least one argument `?argument` and needs to be part of a class `?class` in whose hierarchy `?instvar` is an instance variable. In addition, the method's parse tree is required to match the parse tree of the prototypical setter method implementation shown above. The `setSum(newValue)` method on lines 6–8 in Figure 11 is an example of a setter method that can be successfully detected using this rule since it adheres faithfully to the prototypical setter method implementation.

4.2.2 Fuzzy Logic Rule for Alternative Implementations

To increase the chance of a method being recognized as an alternative setter method implementation, we can once again resort to a heuristic such as the one expressed in the rule near the bottom of Figure 8. This rule does not require a method's parse tree to match the parse tree of the prototypical setter method's implementation, but merely requires that the method contains an assignment expression with the instance variable as its left-hand side and the argument of the method as its right-hand side. To indicate that setter method implementations detected by this rule do not follow the prototypical implementation, we have again annotated the rule with a truth degree of 0.9.

Under similarity-based unification, the rule's final condition will be interpreted as a constraint on the possible values assigned by the assignment statement. The unification of the method's argument `?argument` parse tree node logic term and the parse tree node corresponding to the right-hand side of the assignment expression will succeed with a unification degree of 0.5 if the points-to analysis has determined that both nodes might evaluate to an overlapping set of objects at run-time. Note however that, under similarity-based unification, the left-hand side of the assignment expression might also unify with references aliasing the protected instance variable, possibly introducing false positives.

Table 3 lists the setter method implementations recognized by our fuzzy logic meta programming platform. The `setSum(newValue)` method is detected once with a total truth degree originating from the topmost logic rule in Figure 8. It is also detected once by the logic rule near the bottom with a truth degree of 0.9 using plain unification. Finally, it is detected a third time with a truth degree of $0.45 = 0.9 \cdot \min(1, 1, 1, 1, 0.5)$, originating from the second logic rule in combination with similarity-based unification. The interesting `updateSum(newValue)` method, shown on lines 9–17 of Figure 11, is detected with a truth degree of 0.45 originating from the second logic rule under similarity-based unification. The unification procedure was able to determine that the array indexation expression `arrayOfInts[i]` being assigned to the instance variable `sum`, might alias with the method's argument `newValue`. The `setF()` method depicted in Figure 12, can also be classified semantically as a setter method since it assigns the instance variable `f` the value of its argument `val`.

`if setterMethod(?class, ?method, ?var) : ?c`

?class	?var	?method	?c
SumCmpntVisitor	sum	setSum(newValue)	1
SumCmpntVisitor	sum	setSum(newValue)	0.9
SumCmpntVisitor	sum	setSum(newValue)	0.45
SumCmpntVisitor	sum	updateSum(newValue)	0.45
FooClass	f	setF(val)	0.45

Table 3. Detected setter method instances.

```
correctlyNamedSetterMethod(?c, ?m, ?var) if
  setterMethod(?c, ?m, ?var) : ?c1,
  instanceVariableName(?var, ?varname),
  capitalized(?varname, ?cvarname),
  concat('set', ?cvarname, ?correctSelector),
  selectorOfMethod(?selector, ?m),
  similar(?correctSelector, ?selector) : ?c2,
  ?c1 * ?c2.
```

Figure 9. Logic rule for named setter methods.

`if correctlyNamedSetterMethod(?c, ?m, ?var) : ?t`

?c	?var	?m	?t
SumCmpntVisitor	sum	setSum(newValue)	1
SumCmpntVisitor	sum	setSum(newValue)	0.9
SumCmpntVisitor	sum	setSum(newValue)	0.45
SumCmpntVisitor	sum	updateSum(newValue)	0.2
FooClass	f	setF(val)	0.45

Table 4. Correctly named setter method instances.

4.2.3 Fuzzy Logic Rule for Naming Convention

The Java naming convention for setter methods is the capitalized name of the protected instance variable prefixed by "set". The logic rule for the `correctlyNamedSetterMethod/3` predicate, shown in Figure 9, will find setter methods in an application's source code with an associated truth degree representative for the degree to which each detected pattern instance is well-named. These degrees are listed in Table 4.

4.3 Double Dispatching Methods

Our final use case comprises the detection of the Double Dispatch [3] best practice pattern. The identity of an invoked method depends, in single-dispatching object-oriented languages such as Java, solely on the class of the receiver of a message send. The Double Dispatch best practice pattern is therefore used in cases where application logic not only depends on the class of the receiving object, but also on the class of one of the arguments of the message send. This best practice pattern is, for instance, used in the Visitor Design Pattern [8]. A double dispatching method typically sends a single message to one of its arguments, passing a reference to the current object along.

```
doubleDispatch(?class, ?method, ?message) if
  methodInClass(?method, ?class),
  argumentOfMethod(?argument, ?method),
  variableName(?argument, ?argname),
  methodStatements(?method, ?s)
  ?s = <send(?stype,
             variable(?rtype, ?argname),
             ?message,
             <variable(?vtype, 'this')>)>).

doubleDispatch(?class, ?method, ?message) : 0.9 if
  methodInClass(?method, ?class),
  argumentOfMethod(?argument, ?method),
  expressionInMethod(?exp, ?method),
  isMessageSend(?exp, ?rcvr, ?msg, ?params),
  member(?parameter, ?params),
  thisReference(?method, ?this),
  ?this = ?parameter,
  ?receiver = ?argument.
```

Figure 10. Logic rules for double dispatching.

4.3.1 Logic Rule for Prototypical Implementation

The logic rules shown in Figure 10 capture what it means for a method `?method` to be a double dispatching method in class `?class`, sending a self-reference along using the message named `?message`. The topmost rule corresponds to the prototypical implementation of a double dispatching method shown below:

```
class Foo {
  public void method(Bar arg) {
    arg.methodFoo(this);
  }
}
```

The first two lines of the rule state that a double dispatching method `?method` is a method in class `?class` and has at least one argument `?argument`. In addition, the method's parse tree is required to match the parse tree of the prototypical double dispatching method implementation shown above. The `aceptVisitor(v)` method in Figure 13 is an example of a double dispatching method that can be successfully detected using this rule.

4.3.2 Fuzzy Logic Rule for Alternative Implementations

In order to recognize alternative double dispatch implementations, we resort a last time to a heuristic such as the one expressed in the second rule of Figure 10. This rule does not require a method's parse tree to match the parse tree of the prototypical double dispatching method's implementation, but simply requires that the method contains a message send expression that has a self-reference as one of the parameters and one of the method's arguments as the receiver. To indicate that double dispatching method implementations detected by this rule do not follow the prototypical implementation, we have again annotated the rule with a truth degree of 0.9.

Under similarity-based unification, the unification of the method's argument `?argument` parse tree node and the message `?receiver` parse tree node will succeed with a unification degree of 0.5 in case the points-to analysis has determined that both nodes might evaluate to an overlapping set of objects at run-time. Similarly, the condition `?this = ?parameter` will succeed if the current object of the double dispatching method might alias at run-time with the message `?parameter`.

Table 5 summarizes the double dispatching method implementations recognized by our fuzzy logic meta programming platform. The `aceptVisitor(v)` method of class `Leaf2` is detected once with a total truth degree originating from the topmost logic rule in Figure 10. It is also detected once by the second logic rule with a truth degree of 0.9 using plain unification. Finally, it is detected a third time with a truth degree of $0.45 = 0.9 \cdot \min(1, 1, 1, 1, 1, 1, 0.5, 0.5)$, originating from the second rule under similarity-based unification. The `aceptVisitor(v)` method of class `Leaf1` is also recognized as a double dispatching method although it deviates from the prototypical double dispatch best practice pattern implementation.

`if doubleDispatch(?class, ?method, ?message) : ?c`

?class	?method	?message	?c
Leaf2	aceptVisitor(v)	'visitLeaf2'	1
Leaf2	aceptVisitor(v)	'visitLeaf2'	0.9
Leaf2	aceptVisitor(v)	'visitLeaf2'	0.45
Leaf1	aceptVisitor(v)	'visitLeaf1'	0.45

Table 5. Detected double dispatching instances.

5. Related Work

As our work is an extension of the existing SOUL logic meta programming platform, it is by origin closely related to most of the work SOUL has previously been applied to and which is summarized in Section 3.1. The primary contribution of our extension lies in the end-user transparent way we have incorporated results from advanced static analysis techniques, thus enabling the detection of deviating pattern implementations together with an indication of the platform's confidence in each detected pattern instance. Sections 3.2 and 3.3 contain references to comprehensive introductions to our platform's supporting technologies; being fuzzy logic programming on the one hand and points-to analysis on the other hand.

There is of course a large body of other pre-existing work relying on logic programming for software pattern detection. The PAT [15] system, for instance, extracts the structural relations among classes and methods from C++ header files and stores them as logic facts over which Prolog queries can be launched to find instances of design patterns. It offers, however, no support for reasoning about a method's parse trees, while its reasoning process does not support partial pattern matches nor does it incorporate behavioral information. ASTLOG [4], on the other hand, is a Prolog variant that is particularly well-suited to examining C abstract syntax trees. It avoids the overhead of translating source code into facts by extending the Prolog model such that a goal is always evaluated in the context of a current parse tree node. ASTLOG can be used as an advanced tree walker to locate often recurring bugs using source code templates. It has no support for partial matches nor does it incorporate information about a program's possible run-time behavior.

There is only few existing work incorporating some kind of approximate reasoning in order to detect deviating pattern implementations. Guéhéneuc et al. [10, 9] approach the pattern detection problem from an interesting angle. A constraint satisfaction problem is formulated whose domain covers the application's implementation and whose constraints correspond to the software entities in a pattern's description and the relations between them. The solution to this problem is generated by an explanation-based constraint solver which indicates the constraints that needed to be relaxed in order for a distorted pattern instance to be found, thus identifying shortcomings in the implementation. There are however no approximate inferences as only entire constraints can be dropped from the satisfaction problem. An individual constraint is either completely satisfied or completely dissatisfied. In our approach, each condition in a logic rule can be met with a partial degree of

```java
public class SumCmpntVisitor extends CmpntVisitor {
  private Integer sum;
  public SumCmpntVisitorVisitor() { .. }
  public void visitLeaf1(Component c1) { .. }
  public void visitLeaf2(Component c2) { .. }

  public void setSum(Integer newValue) {
    sum = newValue;
  }

  public void updateSum(Integer newValue)  {
    Integer int1 = new Integer(1);
    Integer int2 = new Integer(2);
    Integer[] arrayOfInts = { int1, int2, int1};
    arrayOfInts[2] = newValue;
    for (int i = 0; i < arrayOfInts.length; i++) {
      sum = arrayOfInts[i];
    }
  }

  public Integer getSum() {
    return sum;
  }

  public Integer returnSum() {
    Integer val = (Integer) indirectReturn(sum, 10);
    return val;
  }

  public Integer retrieveSum() {
    Object retrieved = returnSum();
    if(retrieved instanceof Integer) {
      Integer value = (Integer) retrieved;
      return value;
    } else return getSum();
  }

  public Object indirectReturn(Object o, int delay) {
    if(delay == 0)
      return o;
    else
      return indirectReturn(o, delay - 1);
  }
}
```

Figure 11. Java extract: accessor methods.

```java
public class FooClass {
  public Integer f;
  public FooClass(Integer val) { ... }
  public void setF(Integer val) {
    Object temp = (Object) val;
    CmpntVisitor v = new SumCmpntVisitor();
    Integer z = (Integer) ((SumCmpntVisitor) v).in-
    directReturn((Integer) temp, 15);
    f = z;
  }
}
```

Figure 12. Java extract: a setter method.

```java
public class Leaf2 extends Component {
  public int value;
  public Leaf2() { ... }

  public void aceptVisitor(CmpntVisitor v) {
    v.visitLeaf2(this);
  }
}

public class Leaf1 extends Component {
  public int value;
  public Leaf1() { ... }

  public void aceptVisitor(CmpntVisitor v) {
    System.out.println("Leaf1 accepting visitor.");
    CmpntVisitor tempVisitor = v;
    Leaf1 tempSelf = this;
    tempVisitor.visitLeaf1(tempSelf);
  }
}
```

Figure 13. Java extract: double dispatching methods.

truth which will influence the truth degree of the corresponding solution. This degree of truth is computed according to the laws of fuzzy logic. In Guéhéneuc's approach, weights can be assigned to individual constraints. This gives rise to a preference hierarchy amongst constraints which determines the order in which a problem's constraints are relaxed in order to find imperfect solutions. A metric is derived from these weights measuring the quality of each imperfect solution. Another difference with our approach lies in its inferencing process. It does not incorporate static analysis results about a program's possible run-time behavior. Our ability to detect imperfect pattern implementations does not stem from an automatic relaxation of a rule's conditions, but from a more semantic interpretation of these conditions. By incorporating a similarity-based unification algorithm like the one in LikeLog [2], we could however also present our users an indication of the parse tree nodes that were expected to alias at run-time in order to satisfy the conditions in our logic rules.

Niere et al. [22, 21] propose the use of fuzzy graph rewrite rules for software pattern detection. As having a graph rewrite rule for every pattern implementation variant introduces a large search space, they propose to keep only the commonalities between these graph rewrite rules. Since this introduces false positives and unanticipated correct matches, a weight is added to the resulting rule which represents the anticipated percentage of correct matches. Our fuzzy logic meta programming approach is more flexible as we do not only allow weights to be added to rules, but also incorporate a form of similarity-based unification which results in a more flexible interpretation of the conditions in pattern description rules. Our pattern detection platform depends both on similarity-based unification and on a fuzzified resolution procedure. This way, we can overcome failures in the refutation process caused by a syntactic difference between parse tree nodes which might actually evaluate to overlapping sets of objects at run-time and we are also able to evaluate the confidence our platform has in the discovered software pattern instances.

Jahnke et al. [14, 13] have identified the reverse engineering process as an "imperfect process, driven by imperfect knowledge" given there is a large amount of human involvement required. They therefore advocate that reverse engineering tools should explicitly offer support for modeling uncertainty and contradicting knowledge and have applied possibilistic logic in the reverse engineering of relational database schema.

6. Conclusions

In this paper, we have applied logic meta programming to the validation of an object-oriented program's implementation by ensuring the programs' adherence to a set of user-prescribed software patterns.

In order to detect multiple concrete pattern instances using a single abstract pattern description, we have combined logic meta programming, fuzzy logic and heavy-weight program analysis techniques in a way that is completely transparent to end-users unacquainted with the way static analysis techniques approximate a program's actual run-time behavior.

More concretely, we have modified the resolution procedure to draw sensible conclusions from rules whose conditions are only partially satisfied, while we have altered the unification procedure to take information about a program's possible run-time behavior into account. As a result, our platform is able to interpret conditions in a logic rule as constraints over the run-time behavior source code constructs give rise to instead of as constraints over the literal source code constructs themselves — which is to our best knowledge quite unique.

Our experiments have shown that our prototype implementation is able to recognize, given a software pattern's description as a logic rule, multiple pattern implementation variants giving rise to similar run-time behavior. While the initial experiments described in this paper primarily consist of enforcing software engineering best practices, we are confident that our prototype can be equally applied to some of the more complex logic meta programming applications. In short-term future work, we will for instance investigate its applicability to supporting the co-evolution of an application's design and implementation. On the longer term, we will investigate the use of fuzzy numbers as truth values as well as incorporate other static analyses for which a points-to analysis is a prerequisite.

Acknowledgments

This research was partially funded by the EU-funded network of excellence AOSD-Europe.

References

[1] T. Alsinet. *Logic Programming with Fuzzy Unificiation and Imprecise Constants: Possibilistic Semantics and Automated Deduction*. Spain, Universitat Politécnica De Catalunya, May 2001.

[2] F. Arcelli and F. Formato. Likelog: a logic programming language for flexible data retrieval. In *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC99)*, pages 260–267, New York, NY, USA, 1999. ACM Press.

[3] K. Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1996.

[4] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the 1997 USENIX Conference on Domain-Specific Languages (DSL'97)*, pages 229–242, 1997.

[5] C. De Roover. Incorporating dynamic analysis and approximate reasoning in declarative meta-programming to support software re-engineering. Master's thesis, Vrije Universiteit Brussel, 2004.

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

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

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

[10] Y.-G. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Q. Li, R. Riehle, G. Pour, and B. Meyer, editors, *Proceedings of the 39th Conference on the Technology of Object-Oriented Languages and Systems*, pages 296–305. IEEE Computer Society Press, July 2001.

[11] P. Hájek. Deductive systems of fuzzy logic (a tutorial). Tutorial, 1998.

[12] 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)*, pages 54–61, New York, NY, USA, 2001. ACM Press.

[13] J. H. Jahnke. Cognitive support in software reengineering based on generic fuzzy reasoning nets. *Fuzzy Sets and Systems*, 145(1):3–27, 2004.

[14] J. H. Jahnke and A. Walenstein. Reverse engineering tools as media for imperfect knowledge. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 22–31, Brisbane, Australia, November 2000. IEEE Computer Society.

[15] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, page 208, Washington, DC, USA, 1996. IEEE Computer Society.

[16] R. C. T. Lee. Fuzzy logic and the resolution principle. *Journal of the ACM*, 19(1):109–119, 1972.

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

[18] D. Li and D. Liu. *A fuzzy Prolog database system*. John Wiley & Sons, Inc., New York, 1990.

[19] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Proceedings of the 13th International Software Engineering and Knowledge Engineering Conference (SEKE01)*, 2001.

[20] T. Mens and T. Tourwé. A declarative evolution framework for object-oriented design patterns. In *Proceedings of the International Conference on Software Maintenance (ICSM01)*, pages 570–579, 2001.

[21] J. Niere. Fuzzy logic based interactive recovery of software design. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 727–728, New York, NY, USA, 2002. ACM Press.

[22] J. Niere, J. P. Wadsack, and L. Wendehals. Handling large search space in pattern-based reverse engineering. In *IWPC*, pages 274–, 2003.

[23] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In G. Hedin, editor, *Proceedings of the 12th International Conference on Compiler Construction (CC2003)*, volume 2622, pages 126–137, April 2003.

[24] M. I. Sessa. Approximate reasoning by similarity-based sld resolution. *Theoretical Computer Science*, 275(1-2):389–426, 2002.

[25] T. Tourwé and T. Mens. Identifying refactoring opportunities using logic meta programming. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR03)*, pages 91–100. IEEE Computer Society, 2003.

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

[27] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.