An extensible declarative classification framework and its efficient implementation (Draft)

Petr Jiricka

April 30, 2003

Abstract

We design a generic declarative framework for classification of objects in imperative languages, which allows for arbitrary extensions through callbacks to procedural code. Our work is inspired by some more general tools for knowledge representation, such as prioritized extended datalog programs ([9], [6]). We also outline an efficient and scalable algorithm for evaluating our programs, which is more efficient than evaluation procedures for the more general cases.

1 Introduction and Motivation

Classification of objects based on runtime properties is one of the problems that comes up quite often in the context of application frameworks, such as the NetBeans Integrated Development Environment [19]. For example, it is useful to logically classify files on the disk based on file extension or attributes, the file's content, or other run-time properties, and display a different icon for each file, or provide different user actions based on the file's class. Typically, an object will belong to multiple classifications, each of which captures a different aspect of the object's qualities. Classifications may be related (e.g. some classes may be nested in each other, and some other classes may be disjoint), but they may also be completely independent, or "cross-cutting". In order to be successful, the framework that manages and computes classifications must be powerful enough to capture these concepts.

Additionally, it is desirable that such a framework is declarative, modular, easy to use, performant, scalable, and extensible. It also needs to be tightly integrated with the imperative programming language in which the application framework (IDE in our case) is implemented. In this paper, we aim to design such a framework, and outline the important implementation aspects of such a system.

The rest of this paper is organized as follows: Section 2 introduces *classification programs*, their syntax and semantics, and explains some useful features and properties of the classification framwork on practical examples. Section 3 presents the implementation of the classification framework, including the algorithm for evaluating classification programs, which forms the core of the framework. Section 4 discusses related work. Section 5 discusses future directions. Section 6 concludes the paper.

2 Classification Programs

2.1 Syntax and semantics

This section introduces the *classification programs* and defines their semantics. We start by defining the core syntax of classification programs.

Definition 1 (Classification Program) *A classification program $\mathcal{P}$ consists of* rules *and* preferences. *Rules are of the form*

$$c_0 \leftarrow c_1, ..., c_n, p_1(A_{1,1}, ...A_{1,k_1})..., p_m(A_{m,1}, ...A_{m,k_m})$$

where $n \geq 0$, $m \geq 0$ and each $k_i \geq 0$. c_i are class names, *p_i are* external predicate names *and $A_{i,j}$ are* object constants, *i.e. object instances in the underlying (object-oriented) imperative language*[1]*. We denote the left-hand side of a classification rule the* head *of the rule, and the right-hand side the* body *of the rule.*

Preferences are of the form

$$c_1 < c_2$$

where c_1, c_2 are class names.

We impose an additional constraint. Let $\mathcal{DEP}_\mathcal{P}$ be an oriented graph, whose set of vertices equals to the set of class names appearing in $\mathcal{P}$. There is an edge (c_1, c_2) in $\mathcal{DEP}_\mathcal{P}$ if there is a rule in $\mathcal{P}$ which has c_2 in its head and c_1 appears in its body. Next, let $\mathcal{ORD}_\mathcal{P}$ be a graph constructed from $\mathcal{DEP}_\mathcal{P}$ by adding an edge (c_1, c_2) for for each preference $c_1 < c_2$ in $\mathcal{P}$. The constraint we impose is that we require $\mathcal{ORD}_\mathcal{P}$ to be acyclic.

The intuitive meaning of classification rules is the following: A particular object x belongs to class c_0 (wrt. program $\mathcal{P}$), if at least one of the rules in $\mathcal{P}$ with head c_0 is *satisfied* for x. A rule (in the above form) is satisfied for x if x belongs to classes $c_1, ...c_n$, and all external predicates $p_i (1 \leq i \leq m)$ are true for the tuple of values $[x, A_{i,1}, ...A_{i,k_i}]$. We will explain the idea of external predicates later in this section.

The semantics of preferences will be explained in section 2.2.

The concept of classification rules is closely related to rules in logic programming [17]. Indeed, the rule from definition 1 could be represented by the following logic programming rule:

$$c_0(x) \leftarrow c_1(x), ..., c_n(x), p_1(x, A_{1,1}, ...A_{1,k_1})..., p_m(x, A_{m,1}, ...A_{m,k_m})$$

where x is a variable and c_i, p_j are logic programming predicate names.

[1]We abstract away from the way how object instances are represented in the program text; it is straightforward to represent e.g. string and number constants, and representation for new object types can be added as needed. Also, there are frameworks available which allow for text-based represention of arbitrary object instances.

The notion of external predicates allows for arbitrary extensions of the classification programs. An external predicate is a callback to the underlying object-oriented language, which allows to discover runtime "properties" of the object being classified. In general, the properties may be multi-valued.

An external predicate may be used in two ways: it can either be asked to return all the values of the properties for a given object, or it can be used to check whether the object satisfies a particular value of the properties. There may be performance implications of which of the two modes is used: if the property is multi-valued, returning all values will take longer than determining whether the object satisfies a given (one) value.

Example 1 *Suppose there is an external predicate $representsClass(className)$, which returns Java classes represented by the given Java source file, or checks whether the file represents the given class.*

The object to classify is a file f: `/usr/local/src/com/foo/Product.java`, *which represents java class* `com.foo.Product` *and implements the* `java.io.Serializable` *interface. When queried about all class names for f, the $representsClass$ external predicate will return*

```
[java.lang.Object, java.io.Serializable, com.foo.Product]
```

When queried whether f represents `java.io.Serializable`*, the predicate will return* `yes`*. When queried whether f represents* `java.lang.String`*, the predicate will return* `no`*.*

The idea of external predicates is also inspired by a similar concept in logic programming. For example, the *CommonRules* engine [16] strongly relies on external predicates [2].

In the following, we present some examples of how classes of objects that commonly appear in application frameworks and IDEs can be represented by classification rules.

Example 2 *The class of all Java files can be expressed by the following classification rule:*

$$\texttt{JAVA\_FILE} \leftarrow isFile(), isNotDirectory(), hasExtension(\texttt{java})$$

Here, $isFile()$ and $isNotDirectory()$ are nullary predicates, which determine, respectively, whether an object is a file, and whether it is a non-directory (data) file. Predicate $hasExtension(extension)$ takes one string parameter, and determines the extension of a given file. It is not multivalued, i.e. it always only returns one value for a given object.

[2]The authors of CommonRules use the term *procedural attachment.*

Example 3 *The class of XML files can be expressed by the following rules:*

$$\texttt{XML\_FILE} \leftarrow isFile(), isNotDirectory(), hasExtension(\texttt{xml})$$
$$\texttt{XML\_FILE} \leftarrow isFile(), isNotDirectory(), hasExtension(\texttt{tld}^{3})$$

This example is similar to example 2, but it also demonstrates how classes can be composed using disjunction. Moreover, it illustrates another important point: While it is common knowledge that XML files usually have the `xml` *extension, for* `tld` *files this is less obvious. If the application framework is built in a modular way, then these two pieces of information may be managed by different modules. The basic* XML-module *defines the first rule. The more specialized* TLD-module *can then enhance the class of XML files – "owned" by the* XML-module *– by the second rule. To do this, the* TLD-module *only needs to know the name of this class (*`XML_FILE` *in our case).*

Example 4 *We further enhance the class of XML files by the following rules:*

$$\texttt{JSP\_FILE} \leftarrow isFile(), isNotDirectory(), hasExtension(\texttt{jsp})$$
$$\texttt{XML\_FILE} \leftarrow \texttt{JSP\_FILE}, hasXMLSyntax()$$

In this case JSP stands for "JavaServer Pages", which is another technology used by the Java platform. JSP files can be written using two syntaxes: the "classical" syntax and the XML syntax. Only JSP files written using the XML syntax should belong to the class of XML files. This can be done by defining a class of all JSP files, and then selecting those with XML syntax using the $hasXMLSyntax()$ *predicate. The class of JSP files in XML syntax is a "specialization" of the class of all JSP files.*

This example also illustrates that object can belong to a given class based on various criteria, not just one pre-selected criterion.

The next question to ask is how classes of objects, introduced by classification programs, can be utilitized by the application framework. This was already outlined in section 1: the framework may assign different visual appearance to different classes of objects, or different actions may be available based on object's class(es). We will use the term *feature* for behavior or appearance of objects that is associated with a particular class. Note that the idea of attaching behavior to objects is not a new one – it is the cornerstone of some software design patterns such as the *role object* pattern [13], [3].

We will assume that the application framework defines a set (universe) of features, where each feature is represented by an object in the underlying imperative language. The *feature availability relation* $\mathcal{A}_{\mathcal{P}}$ will denote a binary relation

[3]The `tld` extension is used by the Java platform to identify the so called "Tag Library Definition files". These files follow the XML file syntax.

of $(class, feature)$, which specifies whether a given feature is available for a particular class. This relation is supplied along with the classification program $\mathcal{P}$. So if the framework needs to find all features associated with a particular object x, it must find all classes to which x belongs, and then the features available for these classes. The classification framework and its implementation (see section 3) has been designed specifically to answer such queries very efficiently.

In the next section, we will look more closely at some additional properties of features.

2.2 Ordering of Classes and Features

Every feature belongs to exactly one *feature type.* For example, the feature *"action that opens the object (file) in editor"* belongs to feature type *action.* Or, feature *"icon with resource path `com/foo/bar.gif`"* belongs to feature type *icon.*

Feature types may be *exclusive* or *non-exclusive.* If a feature type is exclusive, then only one feature of a given type may be associated with a particular object. A feature type is non-exclusive, if an arbitrary number of features of this type may be associated with an object. *Icon* is an example of an exclusive feature type, as each object is visualized with the use of one icon only. *Action* is a non-exclusive feature type, as multiple actions may be applicable to an object.

A natural question arises now: How can it be achieved that only one feature of every exclusive feature type is associated with a given object x ? Our framework solves this by introducing ordering on classes to which x belongs.

Definition 2 (Ordering of Results) *Let $\mathcal{P}$ be a classification program and x be an object. Let $C = \{c_1, ..., c_n\}$ be the set of classes containing x wrt. $\mathcal{P}$ (see definition 1). We define a partial ordering $<_c$ on C as the ordering given by the transitive closure of $\mathcal{ORD}_\mathcal{P}$, restricted to C.*

Finally, we define the set of features for each object. This is the output of the classification framework that an application framework may directly use.

Note: In the following, $A \setminus B$ denotes set difference A minus B.

Definition 3 (Set of Features for an Object) *Let $\mathcal{P}$ be a classification program and let x be an object. Let C be the set of classes containing x wrt. $\mathcal{P}$. Let $<_c$ be the partial ordering of C wrt. $\mathcal{P}$. Let $\mathcal{A}_\mathcal{P}$ be the feature availability relation for $\mathcal{P}$. The set of features F_x associated with x is defined as follows:*

- *If f is non-exclusive, then $f \in F_x$ iff $\exists c \in C$ such that $(c, f) \in \mathcal{A}_\mathcal{P}$.*
- *If f is exclusive, then let $C_T \subseteq C$ be the set of classes associated with a feature of the same type as f, and $C_f \subseteq C_T$ be the set of classes associated with f. Now there can be two subcases. If $C_f = \emptyset$, then $f \notin F_x$. If on the other hand $C_f \neq \emptyset$, then $f \in F_x$ iff $\forall c_1 \in C_f, c_2 \in C_T \setminus C_f : c_1 >_c c_2$.*

The intuitive meaning of the above definition is the following. All non-exclusive features are associated with objects that belong to classes for which the given feature is available. For exclusive features, the feature of a certain type may "defeat" other features of that type based on the ordering of classes in the result of a particular object. Since the ordering of classes is partial, a situation may arise that no feature of a given type is selected for a given object, even if the object belongs to classes which are associated with some features of this type. Such a situation should be regarded as an error in the program, and should be corrected by specifying additional preference rules. As we will see later, this situation can be detected by static analysis of the program.

We note that there are two sources of preference information. One is implicit in the classification rules, where classes that are subclasses of other classes are preferred to the superclass. In other words, more specific classes are preferred. This kind of preference has been studied in the context of logic programming, see e.g. [24]. The other source of preferences is explicit, and comes from the preference rules. Also this kind of preference has been studied extensively [6], [12], [7], [23], [5], [25], [15], [1].

In addition to ordering classes in order to prevent any features from being associated with objects which have other incompatible features associated with them, a certain kind of ordering may be useful for one other purpose. We may want to introduce an ordering among features associated with one object. For example, if we are interested in actions that will be presented in the object's visual representation, we want to know in which order the actions should appear in the object's pop-up menu. For this reason, features need to be ordered. We believe that in the context of application frameworks, a sufficient solution is to introduce a global ordering on all features that may ever be associated with any object. This automatically imposes an ordering of features associated with a particular object. The relative order of features will be the same for all objects, but we believe this is appropriate in the context of application frameworks, such as NetBeans.

We close this section by relating some aspects of our classification framework to the general concept of *non-monotonic reasoning* ([18], [11], [10]).

We say that a reasoning (deductive) procedure is *non-monotonic*, if adding new statements to the set of premises may cause some previously drawn conclusions to be withdrawn. We illustrate this on a classic example.

Example 5 *Consider the following facts and deductive rules*[4]*:*

```
Birds normally fly.
Penguins do not fly.
Tweety is a bird.
```

Based on this information, it is reasonable to conclude that Tweety flies. However, suppose that we add a new piece of information:

```
Tweety is a penguin.
```

Now it is appropriate to withdraw the conclusion that Tweety flies, and conclude instead that Tweety does not fly. So adding new information (the fact that Tweety is a penguin) caused a previously drawn conclusion to be withdrawn.

A similar situation may arise in our classification framework in the case of exclusive features. We consider again the program from example 4.

Example 6 *Consider the following classification rules:*

$$\begin{aligned}&\texttt{JSP\_FILE} \leftarrow isFile(), isNotDirectory(), hasExtension(\texttt{jsp})\\&\texttt{XML\_FILE} \leftarrow \texttt{JSP\_FILE}, hasXMLSyntax()\end{aligned}$$

Now assume the following feature availability relation:

$$(\texttt{JSP\_FILE}, \text{editor for JSP files feature})$$
$$(\texttt{XML\_FILE}, \text{editor for XML files feature})$$

This relation basically says that JSP files are edited using the JSP editor, while XML files are edited using the XML editor. In this example, "editor" is an exclusive feature type. Now suppose object x for which $isFile()$, $isNotDirectory()$ and $hasExtension(\texttt{jsp})$ all hold.

From the above information, we would like to conclude that x should be edited using the JSP editor. However, if we add the additional piece of information that $hasXMLSyntax()$ also holds for x, then we want to withdraw that conclusion, and instead deduce that x should be edited using the XML editor.

We have seen that our classification framework can realize some simple forms of non-monotonic reasoning. However, sometimes it would be useful to represent some more complex non-monotonic problems in a classification framework. Section 5 discusses possible enhancements to our framework to make it more general and flexible in the area of non-monotonic reasoning.

3 Evaluating Classification Programs

We now present the algorithm for evaluating classification programs. We are really talking about two algorithms here – one for finding all classes that contain a given object, and another for ordering classes in the result.

3.1 Finding Classes for Objects

The algorithm for finding the classes containing a given object has two phases: the preprocessing phase, and the computation phase. The preprocessing phase is independent of the particular input object, and only needs to be run once (e.g. when the application framework starts up). It is possible to run the preprocessor incrementally and lazily to decrease the initialization time of the framework and to save memory. However, for simplicity of this presentation we will assume that the preprocessing phase is run all at once.

The role of the preprocessing phase it to create a *decision tree*, which then controls the execution of the computation phase. The roles of the control information constructed by the preprocessor are the following:

[4]We assume an intuitive understanding of terms *fact* and *deductive rule*; we do not introduce formal definitions.

- Decide which external predicate to call.
- Based on the results of the external predicate, decide which branch of the decision tree should be taken.
- For the leaves of the tree, specify class names that correspond to the particular leaf.

Given a classification program $\mathcal{P}$, the preprocessor constructs the decision tree starting from the root node, and proceeding towards the leaves. Each node N has an associated program $\mathcal{P}_N$, and a predicate name p_N. The program $\mathcal{P}_N$ is not actually stored in the node, the association is made only for the purpose of this algorithm. Only the predicate name p_N is stored in the node's implementation.

The algorithm starts with an empty tree, containing only the root node N_0. The original program $\mathcal{P}$ is associated with the root node, i.e. $\mathcal{P} = \mathcal{P}_{N_0}$.

In the following, the occurrence of predicate p in a program will be denoted by $p(v)$, where $v = [A_1, ..., A_n], n \geq 0$ is a vector of constants. For each node N (including the root node N_0), the algorithm performs the following steps:

1. Chooses the associated predicate p_N from the predicates that appear in the associated program $\mathcal{P}_N$. If $\mathcal{P}_N$ does not contain any predicates, algorithm proceeds to step 4. Otherwise, choose p_N based on the following criteria. The goal is to choose p_N in such a way that that p appears in as many rules as possible, and the occurrences of p in $\mathcal{P}$ are as diverse as possible, with many different argument values. Also, it is better to choose single-valued than multi-valued ones.

   There may be many different reasonable heuristics for choosing the best predicate, so we don't prescribe a particular implementation.

2. Let $p_N(v_1), ..., p_N(v_m)$ be the occurrences of p_N in $\mathcal{P}_N$. Let $V = \{v_1, ..., v_m\}$ denote the set of values of arguments of these occurrences. Then node N will have 2^m child nodes, corresponding to all subsets of V. Note that if the tree is constructed lazily, so that only the child nodes that are needed for classifying concrete objects are created, the practical space complexity will be much lower. Also, this number of child nodes only applies to the case of multi-valued external predicates. In the case of single-valued external predicates, there will be a maximum of $m + 1$ child nodes, corresponding to subsets of V of cardinality 0 or 1.

   Let U be a subset of V and N_U the child node of N corresponding to U. Child nodes should be implemented as a keyed map, where the key is the particular subset U, and the value is the child node itself. The map should be implemented using some fast mechanism, such as hashing or tries [20].

3. For each child node N_U of N, construct the associated program $\mathcal{P}_{N_U}$. $\mathcal{P}_{N_U}$ is constructed from $\mathcal{P}_N$ as follows:

   - Delete from $\mathcal{P}_N$ all rules which contain an occurrence $p_N(v)$, where $v \in V \setminus U$.
   - Delete from the remaining rules of $\mathcal{P}_N$ all occurrences of $p_N(u)$, where $u \in U$.

    - Delete from the remaining rules all *unreachable* rules, i.e. rules that contain a class in their body which does not appear in the head of any rule. Repeat this sub-step until there are no unreachable rules.

4. If $\mathcal{P}_N$ for some N does not contain any predicates, then N will be a leaf node. It will contain information about classes corresponding to this computation path. For a predicate-less program, the set of classes that form the result of the program is computed by a bottom-up fixpoint computation:

    - Initially, the set of results C is empty.
    - Add to C all classes for which there is a rule $\mathcal{R}$ which has C in its head, and where all classes in the body of $\mathcal{R}$ are already in C. Repeat this step until no new classes can be added.

    This set of result classes will be associated with the leaf node N.

Note: The concept of predicate occurrence $p_N(v)$ is quite straightforward for predicates with at least one parameter, but less obvious for nullary predicates. In such a case, it is important to realize that nullary predicates only have possible resulting sets of values: $\{$`[]`$\}$ (meaning `yes`) and $\emptyset$ (meaning `no`). Thus each such node N will have two child nodes, corresponding to `yes` and `no`. Step 3 of our algorithm for computing the new program for child nodes will collapse to the following: In the `yes` child node, occurrences of p_N are deleted from all rules, whereas in the `no` child node, all rules containing p_N are deleted.

It is now straightforward to see the algorithm for the second phase of the computation, when we already have a concrete object, for which its containing classes should be computed. The computation traverses the decision tree, starting at the root. At each node, it executes the external predicate p associated with this node. The predicate will return a set of values S (which are really vectors of objects). The intersection S_0 of S and the set V of all values of arguments of occurrences of p is used as the key to determine the child node in which the computation will continue. (Remember that child nodes are keyed by the subsets of V.) So the computation will continue in exactly one node, which corresponds to S_0. The result of the computation is the set of class names at the leaf node that the computation entered.

A minor modification of the second phase of this algorithm could be the following. If the size of the set of occurrences V for a particular predicate p_N is small (e.g. $|V| \leq 2$), then it may be faster to query p_N for all the individual values $v_1, ..., v_m \in V$, rather than asking p_N for all values for the given object x. That is because the set of all values of p_N for x could be larger than $|V|$. This will improve the performance of the query especially near the leaves of the decision tree.

An additional performance gain could be achieved in the context of application frameworks by the following. It is quite usual that the framework is only interested to know features *of a given type* t for a particular object (wrt. program $\mathcal{P}$). Then it can:

- Determine the set C_{t_0} of classes that are associated with some feature of type t, i.e. $\exists c \in C_{t_0}, \exists f$ of type t such that $(c, f) \in \mathcal{A}_{\mathcal{P}}$.

- Define the set C_t by expanding C_{t_0} to all classes on which the definition of classes from C_t depends wrt. $\mathcal{P}$.
- Define the program $\mathcal{P}_t \subseteq \mathcal{P}$, containing rules from $\mathcal{P}$ which have classes from C_t in the head. The set of preferences will remain unchanged. This is the program needed to determine the set of features of type t.
- Compute the set of features of type t with respect to the new program $\mathcal{P}_t$.

This program will be more efficient (or as efficient, in the worst case) to evaluate than the original program $\mathcal{P}$, as it contains fewer rules, and thus fewer potential external predicates that may need to be called.

The performance and scalability advantages of our algorithm for evaluating classification programs are easy to see. Each external predicate is only evaluated once, and the number of external predicates evaluated is minimized by choosing those that most limit the search space. This is a much more scalable approach than e.g. the current Datasystems API (see section 4), which may evaluate the same procedural code multiple times.

3.2 Ordering of Classes

Similarly to the preprocessing phase of the algorithm for computing the set of classes containing an object, the algorithm for ordering the classes is independent of a particular object, and can thus be computed only once for each classification program. The algorithm directly follows from the definition of the $\mathcal{ORD}_{\mathcal{P}}$ relation – it is basically a standard graph algorithm.

The class order is associated with each leaf node of the decision tree, so the classes returned at each leaf are in the desired order.

4 Related Work

First, we compare our framework to the current NetBeans mechanisms for object classification. Datasystems API[5] is the legacy procedural NetBeans API used for classifying objects. This API suffers from a number of drawbacks. First, each object can only be assigned to one class by the Datasystems API. So instead of associating the object with both less specific and more specific classes, only the most specific class is considered, and this class must be associated with all the features appropriate for the less specific classes. The concept of cross-cutting classification cannot be realized using Datasystems at all.

Another drawback is its poor performance and lack of scalability, caused by the procedural nature of the API. In essence, in order to determine the incidence of an object to its class, all classes registered for the API must be queried as to whether they contain the given object.

One advantage of Datasystems is that arbitrary classes can be represented, as the procedural classification can run arbitrary Java code.

The NetBeans MIME Resolver API improves on Datasystems by a more declarative specification (allowing faster and more scalable computation), and

[5]Application Programming Interface

by allowing class hierarchy based on specificity. It is still unable to express cross-cutting classes, though.

We believe that our classification framework removes all the above-mentioned drawbacks of these previous approaches.

Our framework is closely related to logic programming [17]. Many motivations for our framework are related to motivations for knowledge representation and non-monotonic reasoning [18], [11], [10].

From the implementation point of view, so far the fastest and most scalable procedures for evaluating logic programs are the top-down procedures based on tabling [4], [8], implemented in the XSB system [22]. We believe that our more specialized framework can outperform the more general XSB system.

The notable idea of *call abstraction* [21], as implemented by the XSB system, is also embodied in our classification framework. Call abstraction refers to the following situation which may arise in a top-down Prolog evaluator. The evaluator needs to execute a given logic programming *goal*, i.e. a predicate call with concrete values or variables. However, it instead executes a more general (abstract) goal, in which some concrete values have been replaced by variables. After the values for this more general goal are returned, it remembers them, so they can be reused by a potential different goal which is also a special case of the general goal that was actually executed. In our framework, the fact that external predicates are queried for all results can be seen as a counterpart of the call abstraction mechanism.

5 Future Directions

There are a number of directions in which the framework can evolve. From the standpoint of application frameworks, a useful feature would be to listen on changes of the classified objects: when the value of a property recognized by an external predicate changes, the set of classes containing the object (and thus the set of features for this object) may change too. We believe that listening on changes could be easily accomodated within the existing framework, by allowing external predicates to listen on changes in objects, and then dispatching such events to the clients of the framework. Also, such an enhancement could be implemented in a very memory-efficient way.

Another useful feature would be to allow transformations of objects. Different external predicates may expect different types of objects, and sometimes it is possible to transform the object to the desired type. So an external predicate could be used for objects it was not primarily intended for. We believe that this feature would also be easy to accomodate.

Another useful direction would be to enhance the non-monotonic reasoning capabilities of our framework. This is usually done by allowing negation, or two kinds of negation [2], [14]. The usefulness of such an enhancement is illustrated by the following example.

Example 7 *If we enhance our language to allow* explicit negation *"$\neg$" [14],* priorities among (labelled) rules *[6], [12], [25] and* defeasible provability *"$\Leftarrow$" [1], then we can write the following program:*

$$\begin{aligned} &r_1 : \texttt{EDITABLE\_IN\_EDITOR} && \Leftarrow isFile(), isWriteable(), isTextFile() \\ &r_2 : \neg\texttt{EDITABLE\_IN\_EDITOR} && \Leftarrow isGeneratedFromUML() \\ &r_2 > r_1 \end{aligned}$$

Rule r_1 can be interpreted as follows: if an object is a writeable text-based file, then unless it can be shown otherwise*, it is editable in the editor. Similarly, rule r_2 means that if a file is generated from UML, then* unless it can be shown otherwise*, it is not editable in the editor. Since rule r_2 has a higher priority than r_1, we want to conclude that if an object satisfies the body of both rules, it is not editable in the editor. This example can not be expressed using the unenhanced framework.*

The area of non-monotonic enhancements presents a problem mainly from the standpoint of efficient evaluation – non-monotonic evaluation procedures are inherently more complex than monotonic ones, or those that only make a limited use of non-monotonic features, such as our framework. Still, limited use of negation (e.g. only on external predicates) seems quite easy to accomodate.

Finally, enhancing the syntax of classification programs to include full Datalog programs would significantly enhance the expressive power of the framework. It would be useful not only for classifying objects, but also for some more general knowledge representation tasks.

6 Conclusion

We have presented a framework for declarative classification of objects in imperative object-oriented languages. The framework introduces a simple language for defining classes, and a mechanism for making callbacks to the underlying imperative language to gather values of objects' properties. The framework allows multiple cross-cutting classifications.

The second part of the framework allows attaching features to objects, including a mechanism to mutually exclude incompatible features through (implicit) class specificity and explicit preference rules. We also outline how features can be ordered, if necessary.

Next, we have presented an efficient and scalable algorithm for evaluating classification programs, which amounts to finding all classifications in which an object is included, and their associated features.

Finally, we have shown the practical use of the framework on a number of problems which frequently arise in the context of modular and extensible application frameworks.

Acknowledgements

The author would like to thank Jesse Glick for interesting comments which helped to improve the quality of this paper.

References

[1] Grigoris Antoniou, David Billington, Guido Governatori, Michael J. Maher, and Andrew Rock. A family of defeasible reasoning logics and its implementation. In Werner Horn, editor, *ECAI 2000. Proceedings of the 14th European Conference on Artificial Intelligence*, pages 459–463, Amsterdam, 2000. IOS Press.

[2] Krzysztof R. Apt and Roland N. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[3] Dirk Bäumer and et al. The Role Object Pattern, 1997.

[4] Roland N. Bol and Lars Degerstedt. Tabulated Resolution for Well Founded Semantics. In *International Logic Programming Symposium*, pages 199–219, 1993.

[5] G. Brewka and T. Eiter. Prioritizing Default Logic: Abridged Report, 1999.

[6] Gerhard Brewka. Well-Founded Semantics for Extended Logic Programs with Dynamic Preferences. *Journal of Artificial Intelligence Research*, 4:19–36, 1996.

[7] Gerhard Brewka and Thomas Eiter. Preferred Answer Sets for Extended Logic Programs. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *KR'98: Principles of Knowledge Representation and Reasoning*, pages 86–97. Morgan Kaufmann, San Francisco, California, 1998.

[8] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[9] Jürgen Dix and Gerhard Brewka. Knowledge Representation with Logic Programs. Technical Report 15–96, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1996.

[10] Jürgen Dix, Ulrich Furbach, and Ilkka Niemelä. Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations. Technical Report 20–98, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1998.

[11] Jürgen Dix, Luis Moniz Pereira, and Teodor Przymusinski. Non-monotonic Extensions of Logic Programming: Theory, Implementation and Applications (Proceedings of the JICSLP '96 Postconference Workshop W1). Technical Report 17–96, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1996.

[12] Thomas Eiter, Wolfgang Faber, and Nicola Leone. Computing Preferred and Weakly Preferred Answer Sets by Meta-Interpretation in Answer Set Programming.

[13] Martin Fowler. Dealing With Roles. http://www.martinfowler.com/.

[14] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[15] B. Grosof. Compiling Prioritized Default Rules Into Ordinary Logic Programs, 1999.

[16] IBM CommonRules. http://www.alphaworks.ibm.com/tech/commonrules.

[17] J. W. Lloyd. Foundations of Logic Programming. Berlin: Springer, 2nd Edition, 1987.

[18] J. Minker. An Overview of Nonmonotonic Reasoning and Logic Programming. Technical Report UMIACS-TR-91-112, CS-TR-2736, University of Maryland, College Park, Maryland 20742, August 1991.

[19] NetBeans IDE. http://www.netbeans.org/.

[20] I. V. Ramakrishnan, Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming*, 38(1):31–54, 1999.

[21] Prasad Rao, C. R. Ramakrishnan, and I. V. Ramakrishnan. On the Optimality of Scheduling Strategies in Subsumption-based Tabled Resolution. In *IJCSLP*, pages 310–324, 1998.

[22] Konstantinos Sagonas, Terrance Swift, David S. Warren, Juliana Freire, and Prasad Rao. The XSB System Version 2.2 Volume 1: Programmer's Manual.

[23] Chiaki Sakama and Katsumi Inoue. Prioritized logic programming and its application to commonsense reasoning. *Artificial Intelligence*, 123(1-2):185–222, 2000.

[24] Frieder Stolzenburg, Alejandro J. García, Carlos I. Chesñevar, and Guillermo R. Simari. Introducing Generalized Specificity in Logic Programming. Technical Report 4–2000, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2000.

[25] Xianchang Wang, Jia-Huai You, and Li-Yan Yuan. Nonmonotonic Reasoning by Monotonic Inference with Priority Constraints. In *Non-Monotonic Extensions of Logic Programming*, pages 91–109, 1996.