Induced Intentional Software Views

Tom Tourwé [a] Johan Brichau [a,1] Andy Kellens [a] Kris Gybels [a,1]

[a] *Programming Technology Lab*
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel
Belgium

Abstract

Evolving and maintaining software requires adequate documentation of its implementation. However, due to the software's constant evolution, the documentation and implementation do not remain synchronised. *Intentional software views* have been proposed as a documentation technique to alleviate this problem. Creating such views is not at all a trivial task, however. In this paper, we propose to use a learning algorithm that derives such intentional software views from *extensional software views*, which are much easier to build. The resulting approach combines the advantages of intentional software views with the ease of constructing extensional views.

1 Introduction

Maintaining and evolving software applications is a complex task, because developers need to understand the software's internal structure to ensure the changes they apply do not break the intended program behaviour (van Gurp and Bosch, 2001). However, this can only be ensured if the design structure is adequately documented. Unfortunately, such documentation is often not available. Even if it was carefully created in the initial development phase, it often becomes outdated over time, because it is not updated when the software evolves. This clearly complicates future evolutions of the software.

Email addresses: tom.tourwe@vub.ac.be (Tom Tourwé), johan.brichau@vub.ac.be (Johan Brichau), akellens@vub.ac.be (Andy Kellens), kris.gybels@vub.ac.be (Kris Gybels).

[1] Research assistant of the Fund for Scientific Research, Flanders, Belgium (F.W.O.)

The major reason for documentation growing outdated is that the software's documentation and its implementation are completely separated. Currently, the software's structure is documented using various diagrams and models (Fowler and Scott, 1999) that are in no way connected to the implementation. Consequently, whenever developers change the implementation, they should remember to update the documentation. Although some documentation techniques that are actively part of the development exist (e.g. SUnit (Beck, 1999)), they also need to be updated manually after each change to the source code (Pipka, 2002).

Extensional and *intentional* software views have been proposed as a way to alleviate this problem. Such views are connected to the implementation and document important structures and patterns (such as design patterns (Gamma et al., 1994)). Intentional views are very powerful, but are hard to construct, because a developer needs extensive knowledge of the internal structure of the software to *identify* an intentional description. Consequently, developers may prefer to use an extensional view, where they can simply enumerate the artifacts belonging to a particular view. Unfortunately, such extensional views are clearly once again not robust towards evolution.

In this paper, we propose an approach that combines the ease of use of extensional views with the advantages of intentional views. To this extent, we use a learning algorithm that automatically derives the intentional view corresponding to an extensional one. Such *induced software views* relieve the developer from the burden of expressing an intentional software view, while it produces documentation that will be updated automatically after evolution.

In the following section, we explain the problem statement in more detail, by discussing the disadvantanges of both extensional and intentional software views. In section 3, we explain the learning algorithm we use to automatically induce intentional software views. Section 4 discusses two experiments we conducted using our approach, and shows the results. In section 5, we generalise these results and analyse our approach. Section 6 discusses related work, while section 7 concludes.

2 Detailed Problem Statement

This section clarifies the problem we want to tackle in this paper. This is achieved by introducing extensional and intentional views, thereby discussing both their advantages and disadvantages.

2.1 Extensional Software Views

Extensional views were introduced by De Hondt (De Hondt, 1998) as *software classifications*. In essence, a view (or software classification) is a collection of source code artifacts, such as classes, methods or variables. In (De Hondt, 1998), De Hondt identifies several *classification strategies*, manual as well as automated, for constructing such views. The most important strategy related to our work is the *manual* classification strategy. It allows a developer to put artifacts in views manually, and is thus the simplest strategy.

Such views exhibit a number of important drawbacks, however, that are all due to the lack of a computable description of the view:

- they are not robust toward evolution, because the view cannot be updated automatically whenever the software changes. Just like ordinary software documentation, the view may thus become outdated after a couple of evolutions;
- they do not scale, because browsing the complete source code and manually classifying artifacts is a time-consuming and error-prone process.
- they do not explicitly mention their underlying intention. A view merely contains some artifacts, but does not explicitly expose the reason why these belong together. Consequently, a developer is expected to discover this intention manually. Naturally, the view may be documented, which provides the developer with some hints of what to look for. This documentation may however also become outdated, and is expressed in natural language, which means it can be interpreted differently by different developers.

2.2 Intentional Software Views

Intentional views are an extension of the extensional views discussed above and were first introduced by Mens (Mens et al., 2002a). Contrary to extensional views, that merely enumerate their items, intentional views provide *a description* of a set of source-code entities that belong to the view. The description is intentional in the sense that the source-code entities in the view are not explicitly enumerated. Instead, all entities belonging to the view are described by means of an executable expression in some programming language. For example, the StarBrowser (Wuyts, 2002) allows the developer to specify views by means of Smalltalk expressions, whereas the Intentional View Browser (Mens et al., 2002a) uses logic predicates instead.

Because an intentional view includes a computable description of its underlying intention, it solves many, if not all, of the problems identified with extensional views:

- they are more robust towards evolution, because the view can be recomputed after the software has changed;
- they scale, because the artifacts are computed automatically, which is essential when dealing with large-scale software systems;
- they explicitly describe the relation between all included artifacts in their intention.

Despite these advantages, intentional views suffer from a number of other problems:

- they are hard to define, since they require meta-programming skills from a developer. Moreover, since their intention is specified in a Turing-complete programming language, it is particularly prone to errors, as are all programs;
- they may include artifacts that do not form part of the view, because the intention specified by the developer is too general. Conversely, the intention may be to restrictive and the view may not include artifacts that should be included. These problems can only be alleviated by the developer, who should adapt the logic rules as appropriate;
- they require extensive knowledge about the software's internal design. Before the intention can be specified, a developer needs to identify it. This is far from a trivial task, however, since browsing a large set of method implementations to reveal some common pattern is a time-sensitive and error-prone process. Apart from this problem, a good rule for describing the intention may not be immediately obvious from the source code and a developer may thus even specify an incorrect rule. Moreover, intentional views confront us with a chicken-and-egg problem: the view should allow a developer to better understand the software, but constructing it already requires a good understanding.

3 Induced Software Views

In this section, we introduce *induced software views* that combine the advantages of both extensional and intentional views, as discussed above. We first present a general overview of this approach. Then, we provide a short introduction to inductive logic programming and logic meta programming, which form crucial cornerstones of our approach.

3.1 General Overview

The approach we propose is actually a combination of the extensional and intentional approach. The idea is that a developer manually classifies the ar-

tifacts he supposes belong together, and that the underlying intention (in the form of the rule that expresses it) between these artifacts is discovered automatically. In this way, we actually combine the ease of use of extensional views, the robustness towards evolution, the scalability and the explicit intention associated with intentional views. At the same time we reduce the major problems associated with intentional views. Since we (try to) discover the intention behind a view in an automatic way, we no longer require the developer to have a deep understanding of the inner workings and structure of the software. Moreover, the rules describing the intention of the view is computed automatically, relieving the developer from the tedious task of writing the program manually, and greatly reducing the risk of errors in the program. We call these learned intentional software views *induced software views*. A summary of the comparison of extensional and intentional views versus induced views is shown in table 1.

View	Classification	Reveals Intention	Evolution	Scalable
Extensional	Enumeration	No	Not robust	No
Intentional	Metaprogram	Yes	Robust	Yes
Induced	Enumeration	Yes	Robust	Yes

Table 1
A comparison of the different software views.

3.2 Inductive Logic Programming

In induced software views, the intention is identified by means of inductive logic programming. This is a machine-learning technique that, given a set of desired example solutions for a predicate, automatically induces rules for that predicate. To derive the rules, an already established set of predicates, called the *background knowledge*, is used. The technique aims to uncover some general pattern in the examples, so that additional examples with the same pattern would be covered by the rules as well. For example, given the following set of examples and background knowledge, the inductive logic program creates the following definition for the `grandFather` predicate [2].

[2] The variables in the induced logic rules have automatically generated names, which we edited for readability.

Examples	Background knowledge	Induced Logic Rules
`grandFather(tom,bob).`	`father(tom,peter).`	`grandFather(?grandfather,?person) if`
`grandFather(tom,jim).`	`father(tom,marie).`	`  father(?grandfather,?father),`
`grandFather(tom,ellen).`	`father(peter,bob).`	`  father(?father,?person).`
`grandFather(tom,bart).`	`father(peter,jim).`	`grandFather(?grandfather,?person) if`
	`mother(marie,ellen).`	`  father(?grandfather,?mother),`
	`mother(marie,bart).`	`  mother(?mother,?person).`

This is achieved using a bottom-up technique (although other techniques exist (Mitchell, 1997)) that generalises a set of specific examples into a more general rule. For example, given the two logic clauses `grandFather(tom,bob)` and `grandFather(tom,jim)`, the technique will automatically deduce the clause `grandFather(tom,?x)`. Because each pair of logic clauses in the examples and the background knowledge is considered in this way, the technique is able to construct a set of logic rules that matches all examples. Since this set of rules contains a lot of redundant parts, a reduction is applied to obtain logic rules as shown above. A discussion of this entire technique, which is called *relative least general generalisation* is beyond the scope of this paper. We refer the interested reader to (Mitchell, 1997).

3.3 Logic Metaprogramming

To use inductive logic programming techniques on software artifacts, we implemented an inductive logic program in Soul (Wuyts, 1998, 2001). Soul is a logic meta programming environment that is implemented on top of, and tightly integrated with, the Smalltalk development environment. The essential distinguishing feature of Soul compared to other logic-based approaches to reason about software (such as (Canfora and Cimitile, 1992)) is that all entities in the object-oriented source code (i.e., classes, methods, variables, inheritance relationships, ...) can be directly accessed from within the Soul environment through a metalevel interface. Some of these predicates are shown in table 2. The main advantage of this approach, as opposed to having a separate repository of logic facts extracted from the code, is that we will always reason about the latest version of the source code, thus avoiding consistency problems. The predicates defined in the metalevel interface will serve as the background knowledge for inducing generalised logic rules.

Soul is a variant of Prolog (Deransart et al., 1996) with some minor syntactic differences. Below we give an example of the syntax. The main differences with Prolog are that logic variables are always preceded by a question mark (e.g., `?P`, `?C`, `?D`) and that the head and the body of a logic rule are separated by `if`, instead of `:-`.

```
classInHierarchyOf(?C,?P) if subclassOf(?C,?P).
```

Logic Predicate	Description
`class(?C)`	`C` must be a class
`subclassOf(?C,?P)`	`?C` must be a direct subclass of class `?P`
`classImplementsMethodNamed(?C, ?M)`	`?C` implements a method named `?M`
`methodSendsMessage(?C, ?M,?S)`	`?S` must be message sent by the method `?M` of class `?C`

Table 2
Predicates in Soul's metalevel interface

```
classInHierarchyOf(?C,?P) if subclassOf(?C,?D), classInHierarchyOf(?D,?P)
```

The logic rules above simply state that a class `?P` is an ancestor of a class `?C` if `?C` is a subclass of `?P`, or if there exists an intermediate class `?D`, which is a subclass of `?P` and an ancestor of class `?C`. Logic queries can be used to trigger the above logic program. For example, the query `if classInHierarchyOf(ScConsExpression,?P)` determines whether a superclass of class `ScConsExpression` exists, and retrieves the result in the variable `?P` (in this case there are two solutions `?P=ScExpression` and `?P=Object`).

4 Experimental Results

In this section, we present two examples of how we used our technique to document the structure of a software application. For each example, we explain the experimental setup, and present the rules that were derived by our induction algorithm.

4.1 Induced View for Scheme Framework

Consider the class diagram in Figure 1, that shows part of a framework for implementing Scheme interpreters (Abelson and Sussman, 1985) in Smalltalk. It depicts two class hierarchies, `ScExpression` and `SpecialFormHandler`. The former is used to represent Scheme expressions, whereas the latter is used to handle the *special forms* defined by Scheme (such as `define`, `if`, etc.). Both hierarchies define a `newClosure` method, that forms part of an instance of the *factory method* design pattern, and that returns an instance of a `Closure` class corresponding to the expression or special form. The idea is to speed up interpretation, by analysing expressions and special forms only once and transforming them into the corresponding closure object, that is then executed as many times as needed. To this extent, the `ScExpression` classes implement an `analyse` method, that analyses all parts of an expression, and initialises the closure object appropriately. Likewise, the `handle:` methods in the `SpecialFormHandler` hierarchy decompose

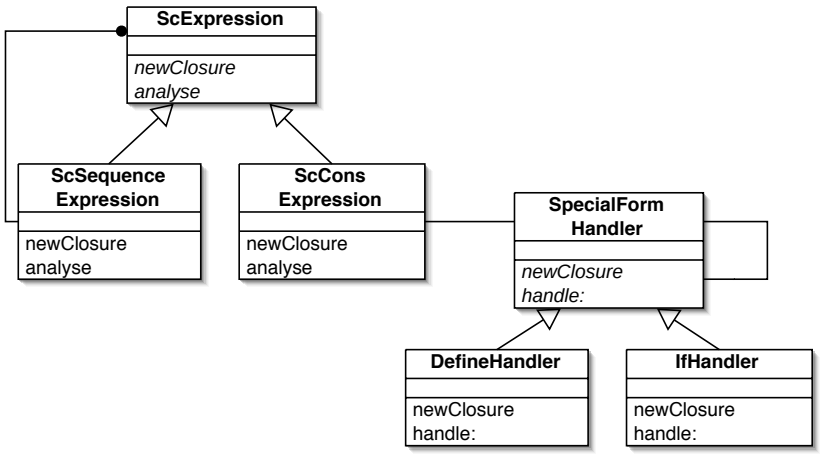

Fig. 1. The `ScExpression` class hierarchy

the special forms, analyse the appropriate subparts and instantiate the closure object correspondingly. Both methods thus form the important part of the analysis phase of the Scheme interpreter.

To document this important behaviour, we want to construct a view containing all methods that belong to the analyser of the Scheme interpreter. The examples we provide to this extent to the inductive logic program are the following.

```
analyser(classImplementsMethodNamed(ScExpression,analyse)).
analyser(classImplementsMethodNamed(ScConsExpression,analyse)).
analyser(classImplementsMethodNamed(ScSequenceExpression,analyse)).
...
analyser(classImplementsMethodNamed(SpecialFormHandler,handle:)).
analyser(classImplementsMethodNamed(DefineHandler,handle:)).
analyser(classImplementsMethodNamed(IfHandler,handle:)).
```

Based on these examples, the inductive logic program produces the following logic rules.

```
(1) intention(analyser,<?class,?selector>) if
       analyser(classImplementsMethodNamed(?class,?selector)).

(2) analyser(classImplementsMethodNamed(?class, handle:)) if
       methodSendsMessage(?class, handle:, newConverterFor:),
       methodSendsMessage(?class, handle:, newClosure),
       methodSendsMessage(?class, handle:, analyse),
       classInHierarchyOf(?class,Scheme.SpecialFormHandler),
       classInHierarchyOf(?class,Scheme.SpecialFormHandlerWithSuccessor),
       classInHierarchyOf(?class, ?class).

(3) analyser(classImplementsMethodNamed(?class, analyse)) if
      methodSendsMessage(?class, analyse, newClosure),
      classInHierarchyOf(?class,Scheme.ScExpression),
      classInHierarchyOf(?class, ?class).

(4) analyser(classImplementsMethodNamed(Scheme.DefineRelHandler,handle:)).
```

The first rule defines the intentional view in terms of the `analyser` predicate,

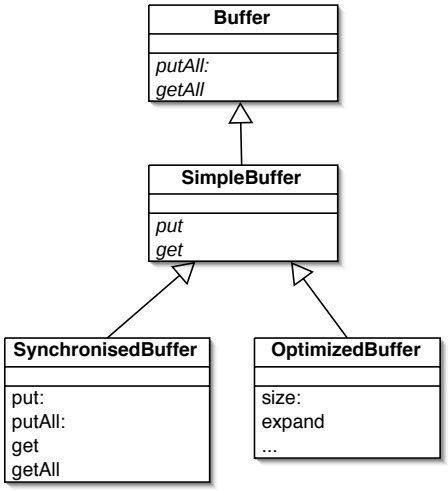

Fig. 2. The `Buffer` class hierarchy

which is defined by the other logic clauses. This `analyser` predicate is defined by two logic rules and one fact that were derived by the induction algorithm. The algorithm has learned that a method is part of the 'analyser' view if:

- it is a method named `handle:` defined in a class in the `SpecialFormHandler` hierarchy that sends the messages `newConverterFor:`, `newClosure` and `analyse` (Rule 2);
- it is a method named `analyse` defined in a class in the `ScExpression` hierarchy that sends the message `newClosure` (Rule 3);
- it is the method named `handle:` in the class `DefineRelHandler` (Rule 4).

Both rule two and three contain some redundant conditions, such as `classInHierarchyOf(?class, ?class)`, which says that the class is part of its own hierarchy, which is always true. This is due to a minor flow in the deduction phase of our algorithm, but does not affect the semantic meaning of the rule.

4.2 Induced View for 'State Changing' Methods

Our second experiment concerns the implementation of an application's update mechanism using the *model-view-controller* (MVC) paradigm (Krasner and Pope, 1988). In our experimental scenario, a developer should incorporate the MVC paradigm into an already existing application. To this extent, he uses software views to document the specific places in the source code where the update mechanism should be invoked. Our goal is then to help the developer in constructing this view and identifying additional places, by deriving the intention behind it automatically.

Figure 2 shows a library of Buffers, implemented as a hierarchy of classes. Buffers are simple data structures that support operations for inserting and

retrieving any kind of elements. Whenever an element is added to or removed from a buffer, its state obviously changes, and this change should be propagated appropriately. In the figure, we show only those methods that are of interest to this experiment, i.e.: the 'state changing' methods that we will classify in a view.

To classify the state-changing methods in a view, a developer needs to identify these methods. He has a number of options to achieve this:

- inspect the source code and look for methods that change the value of some instance variable;
- make use of the coding conventions (e.g. methods prefixed with 'get' or 'set');
- make use of the reflective capabilities of the development environment (e.g. retrieve all methods manipulating an instance variable).

All three options have their respective disadvantages, however:

- inspecting the source code is a time-consuming and error-prone task;
- naming conventions are not always adhered to, so state changing methods risk not to be included in the view;
- the standard reflective capabilities of development environment are limited. More complex queries can be constructed, but require that a developer already knows the intention behind the view, and knows how to meta program.

Because of these difficulties, it is much easier for the developer to classify a number of state changing methods he already identified in an extensional view and then apply the learning algorithm to create a corresponding intentional view. The extensional view can be described by means of the following logic facts.

```
stateChange(classImplementsMethodNamed(SimpleBuffer,put:)).
stateChange(classImplementsMethodNamed(SimpleBuffer,get)).
stateChange(classImplementsMethodNamed(SynchronizedBuffer,put:)).
stateChange(classImplementsMethodNamed(SynchronizedBuffer,get)).
stateChange(classImplementsMethodNamed(OptimizedBuffer,put:)).
stateChange(classImplementsMethodNamed(Buffer,putAll:)).
...
```

Based on these examples, the inductive logic program produces the following logic rules to express the intention of a 'state changing' method:

```
(1) intention(stateChanging,<?class,?selector>) if
        stateChange(classImplementsMethodNamed(?class,?selector)).

(2) stateChange(classImplementsMethodNamed(?class,?selector)) if
        statementInMethod(assign(variable(?var),?expression),?class,?selector),
        instanceVariableIn(?var,?class).

(3) stateChange(classImplementsMethodNamed(?class,?selector)) if
        statementInMethod(send(variable(content),#addFirst:,?expression),?class,?selector).
```

```
(4) stateChange(classImplementsMethodNamed(?class,?selector)) if
        statementInMethod(send(variable(content),#removeLast:,?expression),?class,?selector),
        statementInMethod(return(?expression),?class,?selector).

(5) stateChange(classImplementsMethodNamed(?class,?selector)) if
       statementInMethod(?statement,?class,?selector),
       statementInMethod(send(variable(self),?message,?arguments),?class,?selector),
       stateChange(classImplementsMethodNamed(?class,?message)).
```

These rules express that a method is part of the 'state changing' view if:

- it performs an assignment to an instance variable (rule 2);
- it sends a message `removeLast:` or `addFirst:` to the `content` instance variable (rule 3 and 4);
- it invokes one of the preceding kinds of methods (rule 5).

As these rules show, the learning algorithm is able to derive an intentional view that is much more concise than a view based on naming conventions. Moreover, since these rules are discovered automatically, the developer is not required to know the intention beforehand, nor to implement a corresponding meta program himself.

The view's intention is mostly expressed by rules 2 and 5, that state that a method is state changing if it contains assignments to an object's instance variables or if it invokes one of those methods. However, the derived rules also contain some peculiarities:

- Rule 3 and 4 are too restrictive. First of all, both rules hard code the particular message that is sent to the `content` instance variable, an instance of the `OrderedCollection` class. Clearly, many other methods on this class exist that are also state changing, but these are not considered by the rules. Second, rule 4 states that a method is state changing if a `removeLast:` message is sent *and* a return statement is present. Clearly, this last condition is incorrect. It occurs in the rule's condition because all classified methods that send the `removeLast:` message also contain a return statement;
- rule 5 contains an insignificant first condition, which considers any statement in the method's implementation. Clearly, this condition does not affect the meaning of the rule. It occurs due to a minor flaw in the reduction part of the learning algorithm.

The first problem occurs because the inductive logic program detects the commonalities between the specific examples provided only, and cannot generalise further. This is a documented limitation of the specific algorithm we have used, and is unavoidable. The second problem can easily be avoided by adapting the reduction part of the algorithm.

5 Discussion

As both experiments discussed in this paper show, the proposed approach reveals promising results and can be used to extract the hidden intentions behind a software application's source code. Kellens (Kellens, 2003) performed some more experiments using the approach, to define rules that detect instances of the *Visitor* and *Factory Method* design patterns. However, besides the promising results reported on in this paper, we also identified several particularities with our implementation of the inductive logic program and our approach in general.

First of all, the algorithm we used to implement the inductive logic program proved to be sensitive to the order in which the examples are classified in the view. Depending on this order, the rules ranged from correct and clear to unclear and insignificant. This is a documented drawback of the algorithm we used, and can be solved by using more advanced algorithms. The GOLEM algorithm (Muggleton and Feng, 1990), for example, does not exhibit this problem.

Second, we observed that a sufficient number of examples should be provided to induce a correct set of rules. If the number is not sufficient, the rules are often too restrictive. [+ Hoeveel in onze twee voorbeelden? En kunnen we iets zeggen over of dat in het algemeen zo gaat zijn? +] Furthermore, the amount of background knowledge that is provided is also significant. If this amount is insufficient, the rules may be too general or too specific to the examples at hand. The latter problem occurs in rule 3 and 4 discussed in the previous section, that hard code particularities of the examples instead of generalising from them. This is a general drawback of all inductive logic programs, in particular, and of all machine learning techniques, in general, however. To alleviate it, we envision an approach were the intentional view is derived semi-automatically, allowing the developer to specify extra knowledge to guide the induction process, whenever necessary.

Third, inducing the intention behind an extensional view proved to be extremely time consuming in our experimental tool that uses the Soul interpreter. Even for the small-scale experiments presented in this paper, we were forced to export all necessary information to an external logic compiler. In this external environment, the two experiments took 30 to 40 seconds to complete, which is acceptable given that our experimental algorithm is written as a logic program itself and much more optimized implementations exist. The reason even our relatively simple inductive logic program is so slow, is that Soul is an experimental interpreter, that is not optimized at all. Replacing the interpreter with a more advanced logic virtual machine will certainly speed up the process.

Last, the performance issue also questions the scalability of our approach on large(r)-scale systems, incorporating huge class hierarchies and many method implementations. Because of the experimental state of our algorithm, no performance tests were done, however, so this issue remains to be investigated. In general, the performance of inductive learning algorithms grows exponentially with the number of examples they are given as input. Our experiments suggest that the number of examples necessary to induce a correct rule is rather small, however.

6 Related Work

De Hondt (De Hondt, 1998) was the first to introduce the notion of *software classifications* to group related software artifacts, as explained in Section 2.1. Mens (Mens et al., 2002a) extends this work by introducing an explicit description of which artifacts belong to the view, as a logic program.

Wuyts (Wuyts, 2001) also addressed the problem of keeping different software artifacts synchronised over the lifetime of the software. As a proof-of-concept, the design, which can be considered as a form of documentation, and the implementation of a software system are considered. Tourwé (Tourwé, 2002) shows how the design of a software system can be documented by means of design patterns, and provides high-level design-pattern specific transformations that evolve the software and automatically update the documentation at the same time. Several other authors, most notably (Murphy et al., 1995; Mens, 2000; Murphy, 1996), have shown how coding conventions, design conventions and even architectural styles, can be documented and checked against the implementation. This allows tools to issue a warning whenever this implementation and the documentation are not synchronised.

Many authors reported on the use of machine learning techniques to support the software engineering process. Most of the work concentrates on building models to predict or estimate properties of the software development process or artifacts. For example, Evett (Evett et al., 1998) uses genetic programming to generate software quality models that can predict the number of faults that will be discovered later in the development process, and Mao (Mao et al., 1998) uses decision trees to build predictive models for the reusability of object-oriented programs. An overview of the use of machine-learning to support software engineering can be found in (Zhang and Tsai, 2002).

7 Conclusion

In this paper, we have shown how induced software views can be used to tackle software documentation problems. Such induced software views document important relationships between software artifacts, make this relationship explicit in terms of an intention, and are robust toward evolution. Induced views are created by simply enumerating some artifacts, as with extensional views, and applying the technique of inductive logic programming to uncover the common underlying relationships between them. Contrary to intentional views, they do not require meta-programming skills or extensive knowledge about the software from the developer. In this way, induced software views thus create a synergy between extensional and intentional software views, that alleviates most of their respective disadvantages, while retaining their most important benefits. We have presented two illustrative examples, that showed promising results and illustrated both the feasibility and usefulness of the approach. Further experiments, with more advanced machine learning techniques, are mandatory to validate the approach further and ensure its scalability in larger-scale systems.

References

Abelson, H., Sussman, G., 1985. Structure and Interpretation of Computer Programs. MIT Press.

Beck, K., 1999. Extreme Programming Explained: Embrace Change. Addison-Wesley.

Canfora, G., Cimitile, A., December 1992. A logic-based approach to reverse engineering tools production. Transactions on Software Engineering 18 (12), 1053–1064.

De Hondt, K., 1998. A novel approach to architectural recovery in evolving object-oriented systems. Ph.D. thesis, Departement Informatica, Vrije Universiteit Brussel.

Deransart, P., Ed-Dbali, A., Cervoni, L., 1996. Prolog: The Standard Reference Manual. Springer-Verlag.

Evett, M., Khoshgoftar, T., Chien, P., Allen, E., 1998. Gp-based software quality prediction. In: Proc. 3rd Annual Genetic Programming Conference.

Fowler, M., Scott, K., 1999. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley.

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

Kellens, A., 2003. Using inductive logic programming to derive software views. Tech. rep., Vrije Universiteit Brussel.

Krasner, G. E., Pope, S. T., 1988. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. Journal of Object-Oriented Programming 1 (3), 26–49.

Mao, Y., Sahraoui, H., Lounis, H., 1998. Reusability hypothesis verification using machine learning techniques: a case study. In: Proc. 13th Int. Conf. on Automated Software Engineering.

Mens, K., 2000. Automating architectural conformance checking by means of logic meta programming. Ph.D. thesis, Departement Informatica, Vrije Universiteit Brussel.

Mens, K., Mens, T., Wermelinger, M., 2002a. Maintaining Software Through Intional Source-code Views. In: Proc. Int. Conf. Software Engineering and Knowledge Engineering. ACM Press, pp. 289–296.

Mens, K., Michiels, I., Wuyts, R., December 2002b. Supporting Software Development through Declaratively Codified Programming Patterns. Journal on Expert Systems with Applications .

Mens, T., Tourwé, T., 2001. A Declarative Evolution Framework for Object-Oriented Design Patterns. In: Proc. Int. Conf. Software Maintenance. IEEE Computer Society, pp. 570–579.

Mitchell, T. M., 1997. Machine Learning. McGraw-Hill International Editions.

Muggleton, S., Feng, C., 1990. Efficient induction of logic programs. In: First Conference on Algorithmic Learning Theory.

Murphy, G., Notkin, D., Sullivan, K., 1995. Software reflexion models: Bridging the gap between source and high-level models. In: Proc. of SIGSOFT 1995, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM Press, pp. 18–28.

Murphy, G. C., 1996. Lightweight structural summarization as an aid to software evolution. Ph.D. thesis, Univeristy of Washington.

Pipka, J. U., 2002. Refactoring in a "test first"-world. In: Proc. Int'l Conf. eXtreme Programming.

Tourwé, T., 2002. Automated support for framework-based software evolution. Ph.D. thesis, Departement Informatica, Vrije Universiteit Brussel.

van Gurp, J., Bosch, J., 2001. Design Erosion: Problems & Causes. Journal of Systems & Software 61 (2), 105–119.

Wuyts, R., 1998. Declarative Reasoning about the Structure of Object-Oriented Systems. In: Proc. TOOLS USA'98, IEEE Computer Society Press. pp. 112–124.

Wuyts, R., 2001. A logic meta-programming approach to support the co-evolution of object-oriented design and implementation. Ph.D. thesis, Departement Informatica, Vrije Universiteit Brussel.

Wuyts, R., 2002. Starbrowser.

Zhang, D., Tsai, J., 2002. Machine learning and software engineering. In: 14th IEEE Internation Conference on Tools with Artificial Intelligence.