Language-Independent Detection of Object-Oriented Design Patterns

Johan Fabry [1], Tom Mens [2]

Programming Technology Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium.
Tel: ++32 2 629 33 06 Fax: ++32 2 629 35 25

Abstract

This paper shows that one can reason at a meta level about the structure of object-oriented source code in a language-independent way. To achieve this, we propose a language-independent meta-level interface to extract complex information about the structure of the source code. This approach is validated by defining a set of logic queries to detect object-oriented best practice patterns and design patterns in two different languages: Smalltalk and Java. The queries were applied to two similar medium-sized applications available for each language, and the results were confirmed by manually investigating the source code and available documentation.

1 Introduction

Language independence allows us to reason about OO programs in a generic way. This is very useful from a research point of view, in order to validate promising research in the context of different, yet related languages. For example, logic reasoning about object-oriented source code has been used in Smalltalk for a variety of purposes, and it is desirable to apply these results in a Java setting as well. Some of the interesting applications include, but are not limited to: detecting coding conventions and design patterns [10], bad smells and refactoring [16–18] and code duplication [4].

Email addresses: `Johan.Fabry@vub.ac.be` (Johan Fabry), `Tom.Mens@vub.ac.be` (Tom Mens).

[1] Funded by a doctoral grant of the Institute for the Promotion of Innovation by Science and Technology in Flanders(IWT)
[2] Post-doctoral fellow of the Fund for Scientific Research - Flanders (Belgium)

Through the use of logic meta-programming, where a logic meta language is used to reason about object-oriented base languages [3,19], we achieved language independence by separating language-independent commonalities from language-specific variabilities. More specifically, we made a clear separation between the language independent logic reasoning kernel and three essential language-specific aspects: the meta-level interface (a representational mapping between the logic meta level and the object-oriented base level), the parse tree representation and traversal, and the language idioms.

This enabled us to write logic rules to reason about the structure of object-oriented programs, independent of the actual object-oriented programming language used at base level. In this paper, we experimentally validate this claim by using the same logic queries to reason about object-oriented source code written in two commonly used, yet significantly different languages, namely Java and Smalltalk. As a case study, we focus on the detection of object-oriented best practice patterns [1] and design patterns [6] in the source code. We used this particular case study because of its practical relevance and the fact that it represents a complex problem that is being addressed by many researchers [2,5,13,11]. Knowledge about which and how many patterns are used in the code can be used for a wide variety of purposes such as program comprehension, quality measurement, framework stability, and so on. Moreover, best practice patterns and design patterns have been described for Java as well as Smalltalk [1,6].

Another motivation for selecting Smalltalk and Java is that both languages have some significant differences:

- Smalltalk uses dynamic typing and dynamic compilation, while Java is a statically typed language with static compilation.
- Smalltalk is a language with a clean syntax and very few programming constructs, while Java contains a lot of different programming constructs.
- Smalltalk has runtime reflection, while Java mainly offers introspection, and only a limited amount of reflection.
- Smalltalk has many different dialects, while Java is more standardized (but also more subject to evolution).

2 The Logic Meta Programming Framework

In this section, we present the SOUL logic metaprogramming system that was initially developed to support logic reasoning of Smalltalk code. We discuss how we extended this application with SOULJava, to allow us to reason about Java programs as well. We also explain to which amount the logic system and the associated logic library needed to be restructured to obtain a language-independent framework.

2.1 SOUL

The SOUL system is essentially a Prolog-like programming language and associated logic inference engine, primarily built as a tool for logic reasoning about Smalltalk code [19]. SOUL is written in, and integrated with, Smalltalk, and merges a logic meta language with Smalltalk as base language. Information about Smalltalk programs can be queried and expressed by means of logic facts and rules at the meta level. SOUL comprises three major modules: the logic inference engine, the meta-level interface, and the logic repository.

The meta-level interface (MLI) is responsible for providing the representational mapping between the logic meta language and the object-oriented base language. The implementation of the MLI is, in fact, a hierarchy of classes, subclassing from an abstract `MLI` class, which declares the available meta-level interface. SOUL is designed to allow reasoning about multiple base-level languages (or dialects), by implementing a specific `MLI` subclass for every new target language (or dialect) as illustrated in Fig. 1.

As explained in [19], the MLI for Smalltalk is realised using the powerful reflective facilities offered by Smalltalk. Briefly put, the MLI directly uses reflection to, for example, ascertain if a given parameter is a class or a method, return all classes in the repository, determine the subclasses of a class, and so on.

Smalltalk method bodies are refied by the MLI as functors representing the corresponding node in the Smaltalk method parse tree. For example, a variable reference to `counter` in a method body corresponds to a functor of the form `variable([#counter])`. The message send `counter set: 0` is represented by the functor `send (variable([#counter]),[#set:],<[0]>)`. This representation of the source code allows for straightforward unification of logic rules. For example, the rule to verify if a node in a parse tree represents a variable reference is as follows: `isVariable(variable(?var))`. A message send is verified by the following rule: `isMessageSend(send(?receiver,?message,?arguments))`.

The logic repository contains all logic predicates available to the reasoning engine. While a number of primitive logic predicates are distributed with SOUL, an additional library of over 500 logic rules, called LiCoR, is provided for reasoning about object-oriented source code. Within LiCoR, a separate layer of predicates is dedicated to querying the base level code by accessing the MLI. For example, one such predicate, shown below, is `class(?c)`, which succeeds if the logic variable `?c` is bound to a class that exists in the object-oriented source code. It is implemented by directly accessing a method in the meta-level interface. If `?c` is bound, we verify whether the given value is indeed a class[3]. If `?c` is unbound, `class(?c)` returns multiple bindings, using the `generate` predicate, which will successively bind `?c` to an element of the collection of all classes returned by the MLI:

```
class(?c)  if  atom(?c), [Soul.MLI current isClass: ?c].
class(?c)  if  var(?c),  generate(?c, [Soul.MLI current allClasses]).
```

2.2 SOULJava

To be able to reason about Java source code, we have implemented an extension to SOUL, called SOULJava. However, this does not only require the implementation of a new `MLI` subclass, but also requires an additional code repository to access Java source code from within Smalltalk.

The Java code repository stores all Java code as parse trees: a Java parser is used to transform Java 1.0 source code and store the corresponding parse trees in the repository. Additionally, the code repository contains methods for querying the contained Java trees, which allows the MLI for Java to reify this source code as logic entities. Note that we did not perform any explicit verification of the source code, such as type checks: if the parser succeeds, we simply add the code to the repository. Also, there is no explicit support in SOULJava for compiling or running Java code. The Java parse trees can be exported as Java source code, which can subsequently be compiled using any Java compiler.

The `JavaMLI` subclass implements the meta-level interface for Java, mostly by calling methods on the Java code repository. This MLI also contains 11 extra, Java specific, methods, mostly dedicated to interface support, such as, for example, the `allInterfaces` and `isInterface:` methods. These are called by corresponding logic rules in an addition to LiCoR.

As for Smalltalk, Java method bodies are also represented as logic functors. However, the Java syntax is significantly more complex, which allows for a much wider range of syntactic constructs. Therefore, we had to decide between restricting the logic parse trees to use only functors that are also present in the Smalltalk parse tree representation, or adding new functors for each Java-specific construct. The former would entail a transformation from Java code into 'Smalltalk-compatible' code, which is non-trivial and makes the parse tree representation somewhat dissimilar to the original code. The latter solution is more straightforward as no transformation is required, but it places the burden of handling these new functors on whomever is reasoning about the

[3] Square brackets are used to execute Smalltalk source code from within logic code.

parse tree. We have chosen for the second option because it keeps the logic representation of the Java parse tree similar to the original source code. This is important because we intend to use SoulJava in a later stage to perform source code transformations solely on Java code.

2.3 Language independent framework

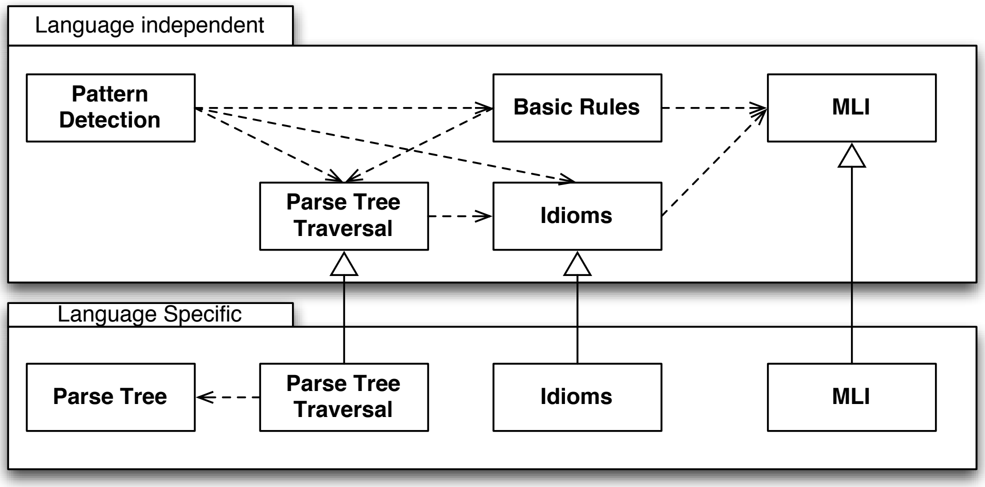

Fig. 1. Separating language-independent and language-specific parts of the logic meta programming system

The fact that we use a different parse tree repesentation for Smalltalk and Java method bodies required us to consider how the rules in the logic library use this parse tree to query the source code. Fortunately, parse tree traversals are handled within LiCoR by a separate *parse tree traversal layer*. Therefore, we needed to provide a second implementation of the parse tree traversal layer of the logic library to support Java. This is illustrated in Fig. 1.

The parse tree traversal layer consists of a number of logic rules, for example, `traverseStatementList(?env,?match,?process,?stats)`, which implements a logic parse tree traversal through the statements of a method. For each node in the tree they verify if the node satisfies the rule given in `?match`. If so, the rule in `?process` will be called with as a first argument the tree node, and as second argument `?env`, to allow the result to be propagated back. If the match was not successful, the subnodes in the tree will be traversed.

Another distinction between languages that we needed to make explicit are so-called language idioms[4], or language-specific naming and coding conventions.

[4] An idiom is a manner of speaking that is natural to native speakers of a language.

For example, the name of an accessor method is equal to the variable name in Smalltalk, and equal to the variable name prefixed with ‘get’ in Java. As shown in Fig. 1, these idioms are implemented as a new *idiom layer* in the logic library. This abstract layer is made concrete for each specific language.

3 Case studies

In order to illustrate the language-independence of our approach we carried out a number of experiments. All the experiments were performed in SOUL version 3.0.15 (with LiCoR version 3.0.14, and SOULJava version 1.2.11) running on top of the Smalltalk/VisualWorks 7.1 software development environment. For each of the selected object-oriented languages (i.e., Smalltalk and Java) we considered two similar applications of which the source code is freely available.

For Smalltalk, we chose *HotDraw* (version 4.5.1 - Sept 2001, 38 classes), a two-dimensional graphics framework for structured drawing editors, and the *RefactoringBrowser* (version 7.1 - Mar 2003, 310 classes) [15], because they are well-known, distributed with VisualWorks 7.1, and documented. This made it easier to validate the results of our experiments. For example, Johnson explains the use of patterns in the HotDraw framework [7]. We chose two very similar applications for Java. *Drawlets* (version 2.0, 115 classes, 23 interfaces, 12 packages), which is the commercial successor of HotDraw for Java. *JRefactory* (version 2.6.38), of which we considered a subset (377 classes, 11 interfaces, 31 packages) is an open source refactoring tool for Java with similar functionality as the Refactoring Browser.

For each of these applications, the presence of a number of best practice patterns [1] and design patterns [6] was detected. We verified our results by investigating the source code and the available documentation.

3.1 Detecting Best Practice Patterns

The best practice patterns we detected were originally proposed for Smalltalk by Kent Beck [1], but are equally valid for Java. In this section, we present the results of our language independent implementation of the following two patterns: *Double Dispatch* and *Getting Method.*

Double Dispatch ([1], page 55) is needed when the behavior of a method depends not just on the class of the receiver, but on the class of one of the arguments as well. When using double dispatch, the implementation of a method consists of returning the result of a single message send to the method’s argument, passing the current object as an argument. Table 1 shows examples taken from HotDraw and Drawlets, with their corresponding logic representations for the parse tree.

Table 1
Examples of double dispatch with corresponding parse tree representation.

	Smalltalk	Java
Class	HotDraw.Figure	drawlets.basic.AbstractFigure
Method	deletionUpdateFrom: aFigure ^aFigure removeDependent: self	boolean isWithin(Figure other) {return other.contains(this);}

Logic parse tree representation	method(HotDraw.Figure,
	  'deletionUpdateFrom:',
	  arguments(
	    <variable([#aFigure])>),
	  temporaries(<>),
	  statements(
	    <return(
	      send(variable([#aFigure]),
	      [#removeDependent:],
	      <variable([#self])>)>)))
		method(
		 drawlets.basics.AbstractFigure,
		  [#booleanisWithin(Figure)],
		  arguments(
		    <variableDecl([#Figure],
		      [#other],[0])>),
		  temporaries(<>),
		  statements(
		    <return(
		      send(variable([#other]),
		      [#contains],
		      <variable([#this])>)>)))

The following logic rule detects double dispatch methods:

```
doubleDispatch(?class,?method,?selector) if
   class(?class), method(?class,?method),
   methodArguments(?method,?args), member(?argument,?args),
   varName(?argument,?argName), varName(?receiver,?argName),
   selfReference(?selfref), member(?selfref,?sendlist),
   or( methodStatements(?method,
         <send(?receiver,?selector,?sendlist)>),
       methodStatements(?method,
         <return(send(?receiver,?selector,?sendlist))>))
```

This rule is language independent, save for the usage of the `varName` and `selfReference` rules. These rules correspond to language-specific idioms, as explained in section 2.3. As shown in Table 2, self references are handled by the language-specific `selfReference` rule, since the name of the pseudo-variable for a self-reference is `self` in Smalltalk, and `this` in Java.

Table 2
Smalltalk and Java specific idiom rules for `selfReference` and `varName`

Smalltalk	Java
selfReference(variable([#self]))	selfReference(variable([#this]))

varName(variable(?name),?name)	varName(variable(?name),?name)
	varName(variableDecl(?t,?name,?d),?name)

The difference in the way variable names are referred and are declared is covered by the `varName` rule. In statically typed languages a variable declaration includes the type of the variable, and a variable reference does not. Therefore, in the logic representation of Java parse trees variable declarations are represented by the functor `variableDecl(?typ,?nam,?dim)`, and variable references by `variable(?nam)`. In dynamically typed languages there is no typing distinction: neither a variable declaration nor a reference includes a type. As a consequence of this, the logic representation of Smalltalk parse trees use the same logic functor for a variable declaration and a variable reference: `variable(?name)`. Considering the code above, the `varName` rule is required to link a variable declaration in the method arguments to the reference of that variable in the arguments of the message send. In other words, viewed procedurally, the `varName(?argument,?argName), varName(?receiver,?argName)` code above first extracts the argument name from the argument declaration and then constructs a variable reference with the argument name.

With the above rules we detected double dispatches on all four selected applications. HotDraw contains no double dispatch method [5] , whereas Drawlets contains three of them (one of which is shown in Table 2). The Refactoring-Browser has 17 double dispatch methods, and JRefactory has 174 of them. Inspecting the last two applications, we observed that almost all of the detected methods are used for implementing a Visitor design pattern, as will be explained in section 3.2.

Getting Methods ([1],page 93-95) A common best practice pattern is the use of getting methods, which simply return the value of an instance variable of the object. In Smalltalk, the name of such a method is equal to the name of the variable whose value is returned. In Java, especially in JavaBeans and Enterprise JavaBeans, the naming convention is to prefix the capitalized variable name with `get`.

Getting methods make the class easier to evolve later on, as the internal representation of the state may be changed without having to modify all users of the class. Usage of getting methods is even required in (Enterprise) JavaBeans where it is the only way instance variables may be accessed.

We can easily detect getting methods using the following rule:

```
gettingMethod(?class,?method,?varname) if
   method(?class,?method),
   methodSelector(?method,?gettingname),
   instVar(?class,?varname),
   gettingMethodName(?varname,?gettingname),
   varName(?var,?varname),
   methodStatements(?method,<return(?var)>)
```

Two idiom rules are used here: `methodSelector` and `gettingMethodName`. As in variable declarations, there is a language-specific difference between the declaration of a method, and the invocation of such a method. In statically typed languages, such as Java, a method declaration includes the return type and the types of the arguments, which are not present when calling a method.

[5] In an earlier version of HotDraw, we detected the double dispatch method shown in Table 2, but it has since been modified.

Applying the `gettingMethod` rule to HotDraw identifies 35 getting methods for the 75 instance variables, while in the RefactoringBrowser 125 getting methods are detected for the 531 instance variables. When looking at the Java applications we see that, surprisingly, this pattern is even less widely used. We detected only 33 getting methods for the 270 instance variables in Drawlets, and 134 getting methods for the 721 variables defined in JRefactory.

Note that it is also common practice in Smalltalk to perform lazy variable initialization, i.e. a variable is initialized upon first access. This is done by extending the getting method with code that first verifies if the variable is uninitialized, and if so, initializes it. If we consider these, we find that HotDraw contains 5 lazy initializers, and the RefactoringBrowser has 128 of them.

3.2 Detecting Design Patterns

We will now discuss how we detected design patterns in a language-independent way. More specifically, we illustrate the *Template Method* and *Visitor* design pattern of [6].

Template Method([6], page 325-330) is a frequently used pattern within object-oriented frameworks: a method in a (usually abstract) class calls abstract methods declared by this class. Subclasses should provide an implementation for these abstract methods, according to the required behavior for that subclass. Template method is one of the main mechanisms for allowing behavior variations in instantiations of a framework. To detect template methods, we obtain a list of all self sends, and return the intersection with the list of all abstract method selectors:

```
templateMethod(?method,?hookSelectors) if
   selfSends(?method,?list), nonEmptyList(?list),
   methodClass(?method,?class),
   abstractSelectors(?class,?AList),
   intersection(?list,?AList,?hookSelectors),
   nonEmptyList(?hookSelectors)

abstractSelectors(?class,?selectorList) if
   findall(?selector,abstractSelector(?class,?selector),?L),
   noDups(?L,?selectorList)

abstractSelector(?class,?selector) if
  method(?method,?class),
  abstractMethod(?method),
  methodSelector(?method,?selector)
```

The `templateMethod` rule indirectly refers three idiom rules: `selfReference`, `abstractMethod` and `abstractSelector`. The first rule is used in `selfSends`, which uses the parse tree traversal to retrieve all self sends in a method. The above rule for the `abstractSelector` predicate is sufficient to cover all occurrences in Smalltalk. In Java, however, a second rule is needed to take interfaces into account:

```
abstractSelector(?class,?sel) if
  methodFromInterface(?class,?meth),
  methodName(?meth,?name),
  or(not(classImplementsMethodNamed(?class,?name,?method)),
     and(classImplementsMethodNamed(?class,?name,?method),
         abstractMethod(?method))),
  methodSelector(?method,?sel)
```

This rule expresses that a class may declare that it implements an interface, which consists of abstract method declarations. When implementing an interface, a class is not required to provide an implementation for all methods declared in that interface. In other words, the class may still contain abstract methods that are not declared in the class itself, but in an interface that the class implements. This is a prime example of a major difference between Smalltalk and Java, yet we see that this is easily handled by one extra rule in the Java-specific idiom layer, which itself uses one idiom rule `abstractMethod` and two rules which call the MLI: `methodFromInterface` and `classImplementsMethodNamed`.

Applying the `templateMethod` rule to the HotDraw Figure hierarchy, only 3 template methods were detected, whereas in the Drawlets Figure hierarchy 42 were detected, 19 of which make use of interfaces. 43 template methods were found in the RefactoringBrowser, and by far the largest amount, 50 methods, were found in JRefactory.

Visitor([6], page 331-344) is extremely useful whenever a variety of operations need to be performed on the same class hierarchy. Instead of implementing the required functionality in each class of the hierarchy, the code for a visitor method is contained within one visitor class. The visited classes use double dispatch to accept a visitor method in the visitor object. This has the advantage that all methods for an operation are contained within one object, and that we can easily add new operations by defining new visitors, without touching the implementation of the tree.

It is a convention to use the words “accept” or “visit” in the method names of the visitor and of the visited nodes of the tree [6]. This makes it easy to detect all tree nodes which can participate in the visitor design pattern by using the following rule, which does not directly use any idioms, but is defined in terms of `doubleDispatch` that relies on the language-specific idioms of Table 2:

```
visitor(?class,?method,?visitorselector) if
   doubleDispatch(?class,?method,?visitorselector),
   methodSelector(?method,?acceptselector),
   visitorNames(?acceptselector,?visitorselector)

visitorNames(?acceptselector,?visitorselector) if
  lowercase(?acceptselector,?acl),
  lowercase(?visitorselector,?vil)
  or(stringContains(?acl,'accept'), stringContains(?acl,'visit')),
or(stringContains(?vil,'accept'), stringContains(?vil,'visit'))
```

As said above, HotDraw contains no double dispatch methods, and therefore, the `visitor` rule does not detect any visitor participants. Similarly, the double dispatch methods in Drawlets are not used for a visitor, so again we have zero results. We have 14 visitor methods in the RefactoringBrowser. and detected 174 visitor methods in JRefactory, for 2 different visitors.

Note that we can encounter false negatives if the naming convention of the visitors is not respected. In all applications we tested, however, the naming conventions were respected. Even if there would have been false negatives, they would have shown up in the list of matches for the `doubleDispatch` rule. Subtracting the results of `visitor` from the results of `doubleDispatch` produces a list of results that can be reviewed for false negatives of `visitor`.

4 Discusion of the results

We analysed all the best practice patterns and design patterns that were detected by our logic rules, and manually cross-checked them with the source code and available documentation. We did not find any false positives, so our approach seems to be reliable. Manual code inspection also did not reveal any false negatives, i.e., occurrences of patterns that were not detected, so our approach seems to be reliable in this respect as well.

The number of design patterns and best practice patterns we reported here is of course very limited. Therefore, further work is needed to apply our approach to other patterns as well. However, the purpose of this research was not to determine new pattern detection rules, or to establish which patterns can be detected through software. We focussed here on patterns for which detection rules have either previously been established or are straightforward. Our aim was to specify these, existing, pattern detection rules in a language-independent manner, showing the feasibility of reasoning about OO software in a language-independent way.

For all the patterns reported on in this paper, the number of language-specific extensions that we needed to make remained fairly small. The meta level interface for Smalltalk required 47 methods, while for Java we needed 58 methods. The logic parse tree traversal for Smalltalk code consists of 14 logic rules, and the traversal for Java consists of 37 methods. Furthermore, for Smalltalk, we needed to implement 6 language idioms, and for Java 8 idioms were required. Extending our experiments to other kinds of patterns, would require new language-specific idioms. However, based on our current experience we believe that the number of idioms that have to be added will remain small.

5 Related work

There are many query tools available that can specify and extract matches in the source code based on *regular expressions* (e.g., the Unix grep tool). These all suffer from the limitation that the queries cannot be used to specify nested and recursive patterns.

[12] described a framework for specifying high-level patterns in terms of programming language constructs. The pattern-matching engine and high-level query language is syntax-driven, so it is not really language-independent, but the ideas seem to be sufficiently generic to be applicable to different languages. Another way to detect programming patterns from source code is by means of software metrics [8].

The FAMIX meta-model [16] also supports multiple OO languages, by means of a *language-independent core* combined with *language extensions* that contain the language-specific details. In contrast to our work, source code is not completely reified: of methods only limited information is kept, such as message sends and variable accesses. This abstracts away from the language-specific details, at the expense of information loss, which therefore limits reasoning capabilities. For example, it is not possible to correctly identify lazy initializers because this requires specifying the control flow, which is not supported in FAMIX.

The research literature also contains many approaches to detect design patterns from source code. Several studies reported in the literature aim at detecting design patterns in object-oriented software based on structural descriptions [13,5] as found in the header files of C++ programs. Unfortunately, they do not take polymorphism into account when identifying pattern-like structures in the code. A more powerful approach, based on the full power of Prolog queries was presented in [9]. Finally, [2] reports on a tool to detect hard-coded design patterns in Smalltalk software.

6 Conclusion

The experiments carried out in this paper showed the feasibility of reasoning about (and more specifically, detecting best practice and design patterns in) OO source code in a language-independent way. We adapted SOUL, an existing logic meta programming system for Smalltalk, into a language-independent framework with two instantiations: for Smalltalk and for Java. The three locations where we needed language-specific extensions were the meta-level interface, the parse tree representation and traversal, and the language idioms.

The meta-level interface provided a representational mapping between the logic meta level and the object-oriented base level. It was defined as a class hierarchy to separate the language-independent commonalities from the language-dependent variabilities. Each of the MLI classes remained relatively small: `MLIforSmalltalk` contains 47 methods, `MLIforJava` contains 58.
The parse tree representation obviously depends on the syntax of the underlying language. A source code parser is required to transform the code to the logic parse tree format. The **parse tree traversal** typically contains one predicate for each kind of parse tree node. We have 14 rules for the Smalltalk tree traversal and 37 rules for the Java traversal. The traversal implementation, however, is very straightforward, and can be automated to a large extent.
The language idioms cover syntactic language differences, naming conventions and coding conventions. For our experiments, we only needed 8 Java-specific and 6 Smalltalk-specific idioms.

The performed case studies have shown us that it is relatively easy to write language-independent detection rules for a selection of best practice and design patterns. One should think in general OO concepts, which has the upside that it avoids getting bogged down in language-specific constructs. Especially when writing rules for OO patterns this is straightforward, as these patterns are usually specified in a language-independent way. Language-specific issues need to be addressed by adding language-specific rules in the idiom layer. This allows us to even cope with large language differences, such as static versus dynamic typing and the availability of interfaces in Java. However, we must keep in mind that we have only detected a subset of all best practice patterns and design patterns, it might well be possible that other patterns are less straightforward to detect. To establish this will require further work.

7 Future work

Obviously, there remain a lot of opportunities for future work:

New patterns. There is a significant amount of patterns for which automated pattern detection is available, and which we have not treated. Interesting future work would be specifying all these patterns in a language-independent way. We can determine which patterns are tricky and which are in fact too language-specific for this approach to be useful.
New languages. Although providing support for new languages, such as C# is non-trivial, the core of the work will be in extending the framework, and not in changes to the detection rules. We believe that adding a new language will not significantly impact the existing detection rules, as they are written in a language-agnostic way.
New language versions. We still need to provide support for Java 1.4, which

will require some work, as we need to rewrite some parts of the parser and of the parse tree traversal.
Reasoning about Java bytecodes. We do not always have access to the source code for Java programs, for example, consider all standard Java libraries. Therefore, it would be very useful to reason about bytecodes as well. Bytecode parsing is not a major hurdle, as we could, for example, simply use an existing bytecode decompiler and reason about the decompiled bytecodes.
Type inferencing. In our current experiments, we didn't use any type information. By resorting to type information, we will be able to provide extra contextual information. Support for type inferencing is again a language dependent issue, since Java has static typing, while Smalltalk uses dynamic typing (for which type inferencers [14] are available).
Code generation. Related to reasoning about software in a language-independent way is generating software in a language-independent way. We are currently investigating how, starting from high-level design descriptions such as UML and pattern descriptions, code can be generated with a minimum of language-specific rules. Initial experiments show that this is indeed feasible, while keeping the code generator clear and concise.

8 Acknowledgments

We thank Kim Mens for feedback during the early stages of writing this paper, the entire SOUL development team, Dirk Deridder for proofreading and Theo D'Hondt for supporting this research.

References

[1] Kent Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, 1997.

[2] K. Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Master's thesis, North Carolina State University, 1996. TR-96-07.

[3] Kris De Volder. *Type-Oriented Logic Meta Programming.* PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 1998.

[4] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proc. Int'l Conf. Software Maintenance*, pages 109–118, September 1999.

[5] L. Cristoforetti G. Antoniol, R. Fiutem. Using metrics to identify design patterns in object-oriented software. In *Proc. Metrics 1998*, pages 32–, November 1998. Bethesda, USA.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems.* Addisson-Wesley, 1994.

[7] Ralph E. Johnson. Documenting frameworks using patterns. *ACM SIGPLAN Notices*, 27(10):63–76, 1992.

[8] Kostas Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proc. Working Conf. Reverse Engineering*, pages 44–54. IEEE Computer Society Press, 1997.

[9] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proc. Working Conf. Reverse Engineering*, pages 208–215, November 1996. Monterey, CA.

[10] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Journal on Expert Systems with Applications*, (23):405–431, November 2002.

[11] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 338–348. IEEE Computer Society Press, May 2002.

[12] S. Paul and A. Prakash. A framework for source code search using program patterns. *Transactions on Software Engineering*, 20(6):463–475, June 1994.

[13] L. Prechelt and C. Kramer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *Universal Computer Science*, 4(12):866–883, December 1998.

[14] P. Rapicault, M. Blay-Fornarino, A.-M. Pinna-Dery, and S. Ducasse. Dynamic type inference to support object-oriented reengineering in Smalltalk. In S. Demeyer and J. Bosch, editors, *ECOOP 1998 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*, pages 76–77. Springer-Verlag, 1998.

[15] D. Roberts, J. Brant, and R.E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[16] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proc. Int. Symp. Principles of Software Evolution*, pages 157–169. IEEE Computer Society Press, 2000.

[17] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *Proc. 7th European Conf. Software Maintenance and Re-engineering (CSMR 2003)*, pages 91–100. IEEE Computer Society Press, March 2003.

[18] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proc. 9th Working Conf. Reverse Engineering*, pages 97–107. IEEE Computer Society Press, October 2002.

[19] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, January 2001.