Ontology-Based Design Pattern Recognition

Damir Kirasić[1] and Danko Basch[2]

[1] Information Support Center
[2] Department of Control and Computer Engineering in Automation
University of Zagreb
Faculty of Electrical Engineering and Computing

2008-09-10

Abstract. This paper presents ontology-based architecture for pattern recognition in the context of static source code analysis. The proposed system has three subsystems: parser, OWL ontologies and analyser. The parser subsystem translates the input code to AST that is constructed as an XML tree. The OWL ontologies define code patterns and general programming concepts. The analyser subsystem constructs instances of the input code as ontology individuals and asks the reasoner to classify them. The experience gained in the implementation of the proposed system and some practical issues are discussed. The recognition system successfully integrates the knowledge representation field and static code analysis, resulting in greater flexibility of the recognition system.

Key words: knowledge-based system, ontology-based system, static code analysis, description logics, OWL application, formal pattern definition

1 Introduction

Design pattern recognition is a part of the code analysis field. As design patterns [1] are descriptions of the *design level*, they may be, and inevitably will be implemented in a number of different ways. Once implemented, design patterns become concrete structures of interacting programming components, and thus difficult to detect.

Discovery of design patterns, and other types of program features, can be used as a basis for diverse objectives like: bug finding [2]; security vulnerabilities discovery [3]; program model checking [4, 5]; program design recovery and reverse engineering [6]; code optimisation [7]; parallelism discovery [8]; software documentation management [9].

A number of design pattern detection techniques have been proposed [14]. Different techniques vary vastly at code and pattern representations, algorithms applied, and overall system architecture. Most of the proposed methodologies are based on static code analysis, but some authors advocate dynamic analysis techniques [10], or combined approach [11]. There is a great variety of approaches to input code representation as well to pattern representations. For example, [12] uses graph-based descriptions that are translated to matrices for both input

code and patterns. The PINOT tool [13] uses an abstract syntax tree (AST), a control-flow graph (CFG) and a data-flow graph (DFG) for code and pattern representations. For a more complete survey and comparison of 26 different techniques see [14].

The purpose of this paper is to describe the ontology-based pattern recognition system. The attribute "ontology-based" refers to the fact that the OWL ontology [15] [16, Chapter 14] forms the core of the system. This ontology precisely and formally defines various code patterns.

The main objectives of the recognition system can be summarised as follows:

- Create an expandable framework for the recognition of various program features;
- Allow for precise (possibly formal) definition of program features that are searched for;
- Separate code pattern description from the rest of the system;
- Create an expandable and easily maintainable pattern ontology;
- Use existing libraries, APIs and proven technologies as much as possible;

The above objectives, actually, list the requirements for the proposed software system. This paper presents the architecture and the design of such a system and describes some experience gained during its implementation.

The recognition system is envisioned as a framework that can be used as a stand-alone utility, or as a subsystem for various larger systems, such as a compiler front end or IDE plug-ins.

The proposed system brings new methodology to the field of static code analysis. It introduces architecture that uses knowledge base as a building block for the recogniser system. Another contribution is in the field of the pattern definition area. To the best of the author's knowledge, there is no pattern detection methodology that uses OWL ontology as a pattern definition medium.

The research project (currently a work in progress) is being done as a part of our Ph.D. research at the University of Zagreb, Faculty of Electrical Engineering and Computing.

The paper is organised as follows. Section 2 presents an overall overview of the proposed system. The following sections describe main modules of the system: parsing and AST generation (section 3), ontologies (section 4), code patterns and recognition procedure (section 5). Some practical problems encountered during the implementation and possible further lines of work are discussed in section 6. The final section reaches a conclusion.

2 Architecture Overview

The overall architecture of the system is shown in figure 1. The main modules of the system are: parser, analyser and ontologies. Viewed as a black box, the whole system takes some source code as input and creates an augmented abstract syntax tree (AST) and analyser reports as output.

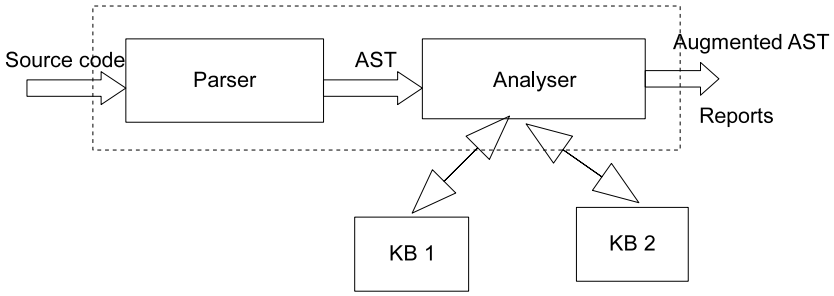

Fig. 1. Architecture overview

Currently, the parser takes the C programming language as input and generates AST of the input program in the form of the XML document [17]. (Section 3 brings more details about parsing and AST generation).

The analyser module takes previously generated AST as input and tries to find code patterns defined in the code pattern ontology. In order to find the specific pattern, the analyser has to follow a sequence of steps that, for complicated patterns, cannot be fully described in the ontology. Therefore, those patterns have to be "known" not only to the ontology but to the analyser code as well. Finally, AST is augmented with additional information and emitted as output. In addition, various reports can be generated. For example, the AST node that represents some program loop can be augmented with information that the loop can be parallelised, etc.

Two ontologies are created for the purpose of the analyser: (i) the programming language ontology and (ii) the code pattern ontology. Both ontologies have been created manually using the Protégé ontology editor [18].

The intent of the architecture presented is to separate the code pattern descriptions from the rest of the system as much as possible for the reasons mentioned in the introduction. Ideally, even the task of the analysis itself should be defined in an ontology.

3 Parsing and AST Representation

The parser module is generated by the ANTLR [20], a well known and widely used language recognition tool. ANTLR is a framework tool for constructing parsers, interpreters and similar programs from language grammars. Along with grammatical rules, the language grammars usually contain actions - code blocks that are executed at the appropriate phase of the parsing process. The ANTLR supports various "target languages". The same grammar can be used for the generation of different parsers - parsers written in a different "target" programming language. C# has been chosen as a target language for the reasons briefly explained later in this section.

The parser generated by the ANTLR is a full-fledged parser that can be used as a compiler front end. It supports full ANSI C grammar and is easily extended

and configured in various ways. For the purpose of this project, C# and Java parsers have been written, but currently are not fully integrated into the rest of the system.

The output of parsing is, as usual, an abstract syntax tree (AST). The generated parsers can create predefined, default AST node types but can be configured to create custom made nodes. By specifying "XML element", as a type for AST node, parser generates a tree of XML elements - actually, a complete XML document. It is not sufficient to specify only the "XML element" as the AST node type but one has to write a custom made class that implements prescribed ANTLR interface.

The parser does full translation. This means that all programming constructs found in the input are preserved and emitted in a different form at the output. For example, for the input code segment:

```
int x = 0;
x = x + 1;
```

the parser will create the following XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<CML version="0.7">
  <!--Created by CToXml 0.7 from 'intx_plus.c'-->
  . . .
  <Declaration>
    <Type>int</Type>
    <DirectDeclarator>x</DirectDeclarator>
    <Initializer>
      <Operator>=</Operator>
      <DecimalLiteral>0</DecimalLiteral>
    </Initializer>
  </Declaration>
  <Statement>
    <AssignmentExpression>
      <Operator>=</Operator>
      <Lvalue>
        <Id>x</Id>
      </Lvalue>
      <Plus>
        <Id>x</Id>
        <DecimalLiteral>1</DecimalLiteral>
      </Plus>
    </AssignmentExpression>
  </Statement>
</CML>
```

It is obvious that the AST (XML document) created is significantly larger then the plain text input code (the actual AST is even larger). But the AST now belongs to the well known realm of XML documents. It is a world with hundreds

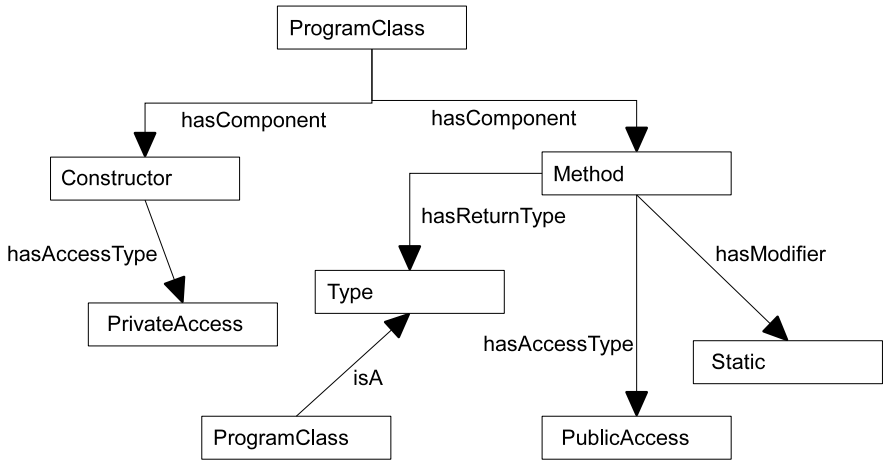

Fig. 2. Simplified programming ontology consists of concepts (boxes) and properties (arrows). A property defines the binary relation between two concepts. Note that `ProgramClass` is a subconcept of a more general `Type`. That is the only concept- subconcept relation shown in figure.

of tested and optimized supporting tools, APIs, parsers, checkers, translators, query engines, etc.

The reason for choosing C# as a target language is that it has powerful XML processing support. Among other things, version 3.5 of C# supports a new lightweight XML document model and the "LINQ to XML" querying constructs [19] that are part of the C# language (not another XML library).

4 Programming and Pattern Ontologies

The purpose of introducing ontologies in this project is manifold: (i) ontology precisely defines *what* has to be found; (ii) ontology brings clear separation of *knowledge* about patterns from the *procedures* of finding them; (iii) ontology brings expandability to the whole system; (iv) ontologies can be easily reused and integrated with other ontologies and other software systems.

For the purpose of the project, two OWL ontologies have been created. The first, called the programming ontology, defines taxonomy of general programming concepts like: Variable, Statement, ForLoop, Type, etc. This ontology contains the OO programming terms as well: ProgramClass, Constructor, Method, AccessType, etc. Figure 2 shows a simplified version of the programming ontology.

The second ontology, called the pattern ontology, defines specific patterns that have to be found.

An example will clarify usage of both ontologies. First, we can assume that the input language is an OO programing language like Java or C#. Second, we suppose that the recogniser has to find all the instances of the singleton classes

[1, 21]. The simplest definition of a singleton is: *"A singleton is a program class that can be instantiated exactly once"*. When this description is translated to code, it inevitably leads to certain program structure (code pattern), that the analyser will try to recognise. Implementing (one version of) a singleton as a Java/C# class, following conditions must be meet: (i) Only private constructors are allowed; (ii) there must be at least one public method (or public static final field) that has the same return type as the class it is defined in; (iii) that public method must return the same instance of the class, no matter how many times called.

The above conditions actually define *restrictions* that have to be applied on a programming class to get the singleton class. Once the restrictions on the program class are clearly articulated, it is not difficult to write a description of the singleton pattern:

```
Class(Singleton partial
   intersectionOf(
   (restriction(hasConstructor someValuesFrom PrivateConstructor))
   (restriction(hasConstructor allValuesFrom PrivateConstructor)))
   ProgramClass)
```

The above description is confined to the first restriction only and defines the `Singleton` as a subclass of the `ProgramClass`. A singleton must have at least one `PrivateConstructor` (in order to avoid automatic constructor generation by compiler) and all of its constructors have to be private. The above code is written in the OWL abstract syntax [15] which is more compact than the official XML OWL syntax.

The second and third restrictions are more complicated and require much more space. Moreover, the second restriction cannot be defined only with OWL but needs SWRL [22] - the rule language that extends the OWL capabilities. Therefore, the reasoner (Pellet [23]), used in this project, must understand both the OWL and the SWRL.

The second restriction has a non-tree structure. We have to state that "the return type of the public static method is the same as the type introduced by the class definition". That kind of non-tree situation is well known problem in the realm of the OWL. (Usually, it is described in terms of `hasUncle` property [22]).

The informal definition (with intuitive meaning) of the rule needed for the second restriction could be written as:

$$ProgramClass(?c) \wedge hasPublicMethod(?c, ?m) \wedge$$

$$hasReturnType(?m, ?c) \rightarrow Singleton(?c)$$

Two approaches are possible now. Either we can try to formulate *all* the restrictions using SWRL rules only, or we can use the OWL to describe as much as we can and then fill the gaps with rules. Adopting the second approach (and assuming that class `SingletonCandidate` complies to the first restriction) the second restriction has to be reformulated to:

$$SingletonCandidate(?c) \wedge hasPublicMethod(?c, ?m) \wedge$$

$$hasReturnType(?m, ?c) \rightarrow Singleton(?c)$$

In order to test the validity of the pattern definition we can define, in the ontology editor, the test individual as an instance of the `ProgramClass` and ask the reasoner to classify [16, 24, 25] all the individuals present in the ontology. If the reasoner finds that all the restrictions are satisfied, it can deduce that the test individual is a `Singleton`. The same logic is followed in the analyser module that is presented in the next section.

5 Finding Code Patterns

The main module of the whole system is the "Analyser" box shown in Fig. 1 section 2. It takes an XML document (AST) as input and augments some of the input nodes according to the patterns defined in the pattern ontology. The analyser can produce additional reports as well. Note that the XML document constructed by the parser does not have to be saved in a file and then be re-parsed again by the analyser. It is held as in-memory tree that is passed to the analyser.

Depending on the type of the pattern, the recogniser will inspect the input tree and try to match input nodes with a specific pattern. In case of the singleton pattern, described in the previous section, it will construct an instance of the `ProgramClass`. The properties of the constructed individual will correspond to the actual properties found in the input tree. Once the construction is finished, the newly constructed instance is added to the pattern ontology. Now the reasoner is asked if the added individual is an instance of the `Singleton` class. If the reasoner answers positively, we have a match.

Other types of patterns can require different scenarios. One of the main objectives of this project is to explore possible ways of pattern description and recogniser-ontology interaction.

The interaction between the analyser and the ontologies is accomplished by the OWL API [26] - open source Java library. The .jar libraries were previously translated from .jar to .dll.

6 Practical Issues and Further Work

Three situations encountered during the implementation phase seem to be important and worth elaborating: (i) non-tree ontology structures; (ii) analyser code most "know" about ontologies; (iii) ontology and reasoner APIs are defined for Java but the project uses C#.

The non-tree structure of the ontology is already shown in Fig. 2 in section 4. The `ProgramClass` has public static method that returns the same type as

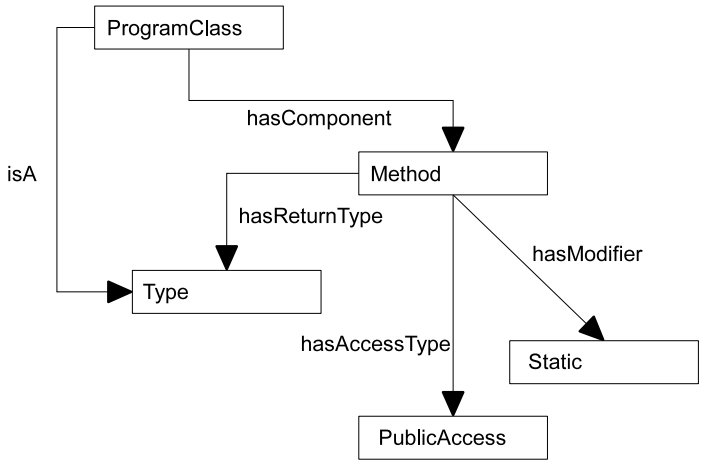

Fig. 3. The non-tree structure from Fig. 2

the class the method is defined in. Therefore, the same relations should be more accurately depicted as in Fig. 3.

The non-tree property of the structure is more obvious in Fig. 3 than in Fig. 2. The presence of non-tree structures were the rationale for the introduction of the SWRL rules. This approach implies the usage of the SWRL-supporting reasoner as well. It would be much cleaner to have pure OWL only.

The analyser must "know" about patterns. For example, the reasoner must know that for the singleton pattern it has to construct a new instance, and that it has to add a newly created instance to ontology, and that it has to ask the reasoner about inherited types etc. It would be very desirable to have these steps defined in ontology as well. Another possibility is to have a uniform, pattern-independent algorithm for *all* pattern descriptions.

Almost all libraries and APIs for ontology processing are written in Java but the project is written in C#, for the reasons mentioned before. Currently, the project uses Java bytecode to MS CIL translator: IKVMC (see http://www.ikvm.net). This tool was used for the translation of huge .jar libraries to .dll libraries. Although no problems were encountered, it would be much more desirable to have native C# libraries for OWL processing

In addition, a more detailed evaluation of this work has to be done. Primarily, comparison with other similar approaches and verification of accuracy (soundness and completeness) has yet to be performed.

7 Conclusion

The proposed system uses an ontology as a basis for pattern definitions. The main points that describe this system can be summarized as follows:

– The parser generates AST as an in-memory lightweight XML tree.

- The AST can be easily and elegantly processed by many tools and APIs (such as MS LINQ to XML).
- The code patterns searched for are formally described in the separate standalone ontology.
- The pattern recognition system can be easily expanded, modified and integrated with other systems.
- The system successfully integrates formal knowledge bases and static code analysis.

Although the pattern recogniser works well, there are many features which can be improved. The most important area of possible improvement seems to be the area of knowledge-algorithm integration.

References

1. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional, (1995)
2. Nick Rutar, Christian Almazan, and Jeff Foster. A Comparison of Bug Finding Tools for Java. In Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering, St. Malo, France, Nov 2004.
3. M. Pistoia, S. Chandra, S. J. Fink and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. IBM System Journal, vol 46, no 2 (2007)
4. Dawson Engler and Madanlal Musuvathi. Static Analysis versus Model Checking for Bug Finding. In Verification, Model Checking and Abstract Interpretation (VMCAI'04), volume 2937 in LNCS, pages 191–210, Jan 2004.
5. Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori and Sriram K. Rajamani. Synergy: A New Algorithm for Property Checking. In FSE '06: 14th Annual Symposium on Foundations of Software Engineering, November 2006
6. J. Niere, W. Schafer, J. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In Proceedings of the 24rd International Conference on Software Engineering, pages 338–348. ACM, (2002)
7. Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar. 2004.
8. David Tarditi, Sidd Puri and Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. Proceedings of the 12th international conference on Architectural support for programming languages and operating systems 2006, San Jose, CA, USA October 21 - 25, 2006.
9. Yonggang Zhang, Ren Witte, Jurgen Rilling, Volker Haarslev. An Ontology-based Approach for Traceability Recovery. 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006), Genoa, October 2006.
10. Janice Ka-Yee Ng and Yann-Gal Guhneuc. Identification of Behavioral and Creational Design Patterns through Dynamic Analysis. In Andy Zaidman, Abdelwahab Hamou-Lhadj, and Orla Greevy, editors, Proceedings of the 3^{rd} International Workshop on Program Comprehension through Dynamic Analysis, pages 34–42, October 2007.

11. Nikolas Pattersson. Measuring precision for static and dynamic design pattern recognition as a function of coverage. International Conference on Software Engineering, St. Louis, Missouri, (2005)
12. N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.Halkidis, Design Pattern Detection Using Similarity Scoring. IEEE transaction on software engineering, Vol. 32, No. 11, November 2006.
13. N. Shi and R. A. Olsson, Reverse engineering of design patterns from Java source code. 21st IEEE/ACM International Conference on Automated Software Engineering, (2006)
14. Jing Dong, Yajing Zhao, Tu Peng: Architecture and Design Pattern Discovery Techniques - A Review. 621-627. H. R. Arabnia, H. Reza (Eds.): Proceedings of the 2007 International Conference on Software Engineering Research & Practice, SERP 2007, Volume II, Las Vegas Nevada, June 25-28, 2007
15. Mike Dean and Guus Schreiber, Editors, OWL Web Ontology Language Reference, W3C Recommendation, 10 February 2004, Latest version available at http://www.w3.org/TR/owl-ref/
16. F. Baader, D. Calvanese, D. McGuinness, D. Nardi and P. F. Patel-Schneider (ed.) The Description Logic Handbook (2nd Edition): The theory, Implementation and Applications. Cambridge University Press, (2007)
17. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Franois Yergeau eds. Extensible Markup Language (XML) 1.0 (Fourth Edition). Available at http://www.w3.org/TR/2006/REC-xml-20060816/. W3C Recommendation 16 August 2006.
18. Holger Knublauch, Ray W. Fergerson, Natalya F. Noy, Mark A. Musen. The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications Third International Semantic Web Conference - ISWC 2004, Hiroshima, Japan, (2004)
19. Joseph C. Rattz, Jr. Pro LINQ: Language Integrated Query in C# 2008. Apress, (2007)
20. Terrence Parr. The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Programmers, (2007)
21. Joshua Bloch: Effective Java Programming Language guide. Addison-Wesley Professional, (2001)
22. Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission 21 May 2004. Latest version is available at http://www.w3.org/Submission/SWRL/.
23. Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur and Yarden Katz. Pellet: A practical OWL-DL reasoner, Journal of Web Semantics, 5(2), (2007)
24. Boris Motik, Rob Shearer and Ian Horrocks. Optimized Reasoning in Description Logics using Hypertableaux. In Proc. of the 21st Int. Conf. on Automated Deduction (CADE-21) Vol. 4603 of Lecture Notes in Artificial Intelligence, pages 67–83. Springer, (2007)
25. Ian Horrocks, Ulrike Sattler. A Tableau Decision Procedure for SHOIQ, J. of Automated Reasoning, Vol. 39, No. 3, pages 249–276. (2007)
26. Matthew Horridge, Sean Bechhofer, Olaf Noppens. Igniting the OWL 1.1 Touch Paper: The OWL API. OWLED 2007, 3rd OWL Experienced and Directions Workshop, Innsbruck, Austria, June 2007.