Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 94 (2004) 19–28
www.elsevier.com/locate/entcs

A Meta-Model for Fact Extraction from Delphi Source Code

Jens Knodel[1] and G. Calderon-Meza[2]

Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6, D-67661 Kaiserslautern, Germany

Abstract

Fact extraction by parsing is often the first step when analyzing a software system in a reverse engineering context. Essential to a fact extractor is the underlying meta-model, which specifies the elements and relations to be extracted. In this work we will introduce a meta-model for the Delphi programming language. The meta-model will be compared against the Dagstuhl middle model (DMM) and reasons for the development of an additional meta-model will be given. Furthermore, we will report on our first experiences with a fact extractor currently under development. We evaluate our fact extractor with two software applications that were developed in the open source community. In particular we give numbers and examples to point out the capabilities and the not yet resolved open issues of our fact extractor and we will reflect our experiences made during the case studies.

Keywords: Delphi, fact extraction, meta-model, parsing, reverse engineering

1 Introduction

Reverse engineering and especially architecture recovery aim at extracting higher-level representations (e.g., the software architecture [6]) directly from the software system (i.e., the source code) in order to support developers in assessing, maintaining, and evolving large-scale software systems [2]. To produce such architectural views current reverse engineering tools process various artifacts available for the system under study such as source code, scenario profiles, documentation, domain information and expert knowledge.

[1] Email: knodel@iese.fraunhofer.de
[2] Email: gcalde@ieee.org

doi:10.1016/j.entcs.2004.01.007

Fact extraction from source code aims at the finding pieces of information about the system (e.g., a fact is that a class is named Class-1 or that function-A calls function-B). It is a fundamental step for reverse engineering techniques and often has to be performed as a first step [4,7,8,11,13]. That means, before performing any high-level reverse engineering analysis or architecture recovery activities, available information in the source code has to be extracted and aggregated in a fact base or repository. Such a fact base then can be the basis for further analysis or recovery tasks.

A common technique for extracting facts from source code is parsing. In this position paper we will introduce a meta-model for the Delphi programming language [3]. We introduce a meta-model for the Delphi programming language and a fact extractor grounded on this meta-model. The fact extractor produces an output format suitable for further analysis, the Rigi standard format RSF [14]). The RSF format is a stream of triplets that follows a "verb subject object" notation. The meta-model and the fact extractor support currently Delphi 5. Once it is stable, we plan to apply our fact extractor in reverse engineering projects to gather information about software systems that use Delphi. To our knowledge, our work is the first attempt to develop a fact extractor targeted especially on Delphi.

The remainder of this position paper is structured as follows: Section 2 gives a short overview about the Delphi programming language and compares it against the well-known programming language C++. Section 3 describes our Delphi meta-model, which underlies our fact extractor. The entities and relations, which we want to extract, are presented there. The fact extractor (although still in development) is validated preliminarily with two case studies in Section 4. Then Section 5 discusses related work by comparing our work against a language independent meta-model, namely the Dagstuhl middle model. Section 6 presents our future work in this area. Finally, Section 7 summarizes our work and draws some conclusions.

2 The Delphi Programming Language

Delphi is a Borland [1] product based on the Pascal language, a third generation structured programming language, initially developed by Niklaus Wirth in the late 1970s. Delphi is targeted to Microsoft Windows and Linux environments and has enhanced object oriented capabilities. The latest version is Delphi 7, but Delphi 5 is still more widespread. Table 1 matches Delphi's programming languages entities to those of C++. Both languages support procedural programming paradigms as well as object oriented concepts. There are several differences:

- C++ has two different types of files: implementation (.c or .cpp files) and header (.h or .hpp files) files. Header files specify interfaces and data structures that can be imported by other files. In Delphi, there is only one type of file, .pas files, which contains sections for interface definition and an implementation, both only separated by special keywords.
- C++ does not have the concept of properties of a class. When used, properties look and act like fields but, in fact, their functionality is implemented by methods. Properties take the place of accessor and mutator methods (often called getter and setter), but have more power and flexibilities:
  A property has a reader and writer to get and set the property's value. Read-only and write-only values that allow only restricted access can be easily implemented by using properties. Properties are used in both ways, like fields and like routines, and their is no performance loss when using properties instead of another concept. For more details on how properties are used effectively see [10].
- In C++ there is no clear distinction between functions (with a return type) and procedures (without a return type) as it is done in Delphi, where a keyword clearly indicates this. In C++, there are only functions, although the return type may be void.

3 The Delphi Meta-Model

The following figures describe the meta-model in UML [16] notation for the Delphi programming language. White nodes stand for Delphi code elements, while the gray ones represent abstract elements. The file system provides a starting point for the meta-model. Figure 1 depicts the physical organization aspects of Delphi's meta-model. Source code files can be located anywhere in the file system. Usually, the source code has one common root directory that can contain several other directories as well as source code files. A file implements at least one unit. A unit can be regarded as a namespace where in every identifier must be unique. We distinguish between Delphi units and system units. Delphi units are application related units implemented by the developers themselves. These units must be analyzed in order to extract information about the application. System units are library or third party units, which are only used by the application. We find it useful to have a clear separation between those two different types of units. Depending on the goals of the later analysis activities, this distinction can bring valuable information into the reverse engineering activities.
Figure 2 shows the different programming languages elements that a unit can contain, namely types, classes, variables, properties, members and fields,

Delphi	**C++**
Implementation file: .pas	Implementation file: .c, .cpp
-	Header file: .h
Unit	Namespace
Class	Class
Type	Struct
Property Of Class	-
Member	Member
Field	Field
Class Function	Class Method
Class Procedure	Class Method
Function	Function
Procedure	Function

Table 1
Delphi versus C++

leaving out routines, which are shown in detail in Figure 3. A unit provides functions and procedures, and contains classes, types and global variables. A class is composed of class routines, fields (i.e., members of a class) and class properties. Each type consists of one or several members.

Meta-model elements are connected to each other via relations. Relations form the verb (what kind of relations exists between subject and object) in RSF triplets. For Delphi, we regard the following relations:

- Call: Routines (i.e., procedures (of classes) and functions (of classes)) can call each other.
- Import: Units can import other units. The content of the imported ones is then visible for the unit.
- Inherits: A class can inherit from another class.
- Level: the contents hierarchy is captured in the level relations, i.e. when element-A contains element-B then we have the relation: level element-A element-B
- Set: A routine is able to set the value of an entity (i.e., members, variables, properties, fields)

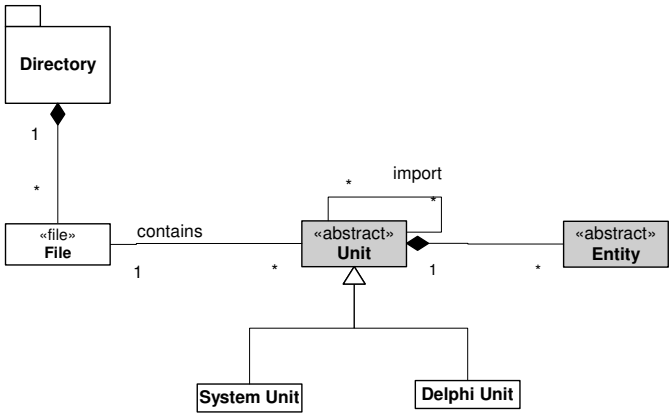

Fig. 1. Files

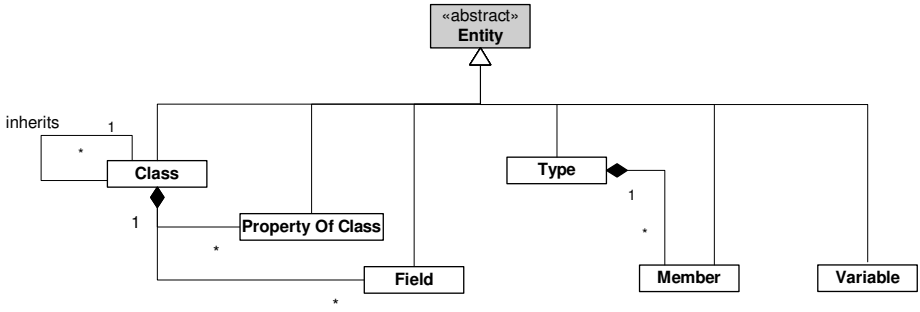

Fig. 2. Entities

- Use: And a routine can use an entity as well.

4 Case Studies

In our case studies, we analyzed a Model Scene Editor (MSE) [12] and a SQL Parser (SQLP) [15] both are open source projects of the Delphi community. MSE is a 3D Scene Editor supporting POVray V3.1 and other formats such as VRML and DirectX. It consists of 113 files and approximately 37 KLOC. SQLP (version 0.01 alpha) is a string parser that is capable to parse SQL statements into tokens, to change these tokens and to rebuild (modified) SQL statement. It consists of 32 files containing about 7 KLOC.

Table 2 gives an overview about the numbers of the meta-model elements, while Table 3 shows the numbers of different relations of the fact base after

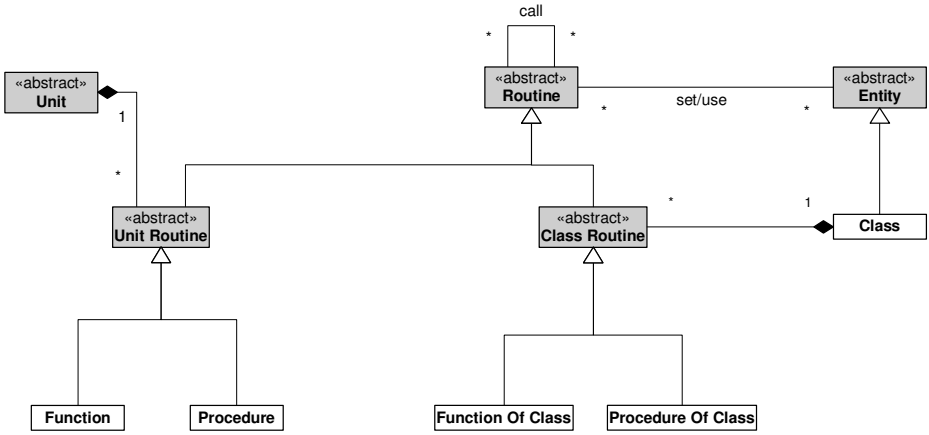

Fig. 3. Routines

Meta-Model Elements	Number of Occurrences	
	MSE	SQLP
Class	325	89
Delphi Unit	113	32
Directory	1	1
Field	3098	211
File	113	32
Function	126	8
Function Of Class	216	141
Procedure	43	5
Procedure Of Class	1025	512
Property Of Class	4	156
System Unit	40	27
Type	418	27
Variable	3681	88

Table 2
Meta-Model Elements

Relations	Number of Occurrences	
	MSE	SQLP
Call	772	587
Contains	113	32
Import	1431	221
Inherits	35	38
Level	9162	1298
Set	405	92
Use	354	207

Table 3
Relation Results

applying our fact extractor. Overall, the output of the fact extractor for the MSE results in 21467 RSF [14] triplets, describing a total of 9201 different meta-model elements, while SQLP has 3807 triplets and 1329 different elements.

When applying the fact extractor to the two open source software systems, we faced the following problems not yet completely solved:

- The fact extractor does not yet cover all features of prior Delphi versions as well as we face the problem that probable future extensions of Delphi.Net may require changes in the implementation of the fact extractor. Imaginable is to build a product family of fact extractors, that share most of the language grammar and functionality like the output and report mechanisms as a common core. Differences between Delphi versions are explicitly captured as variabilities, so that it is possible to derive instances for each needed version.
- Complex expressions containing several calls and uses within a single statement are not yet recognized completely because those expressions require a type analyzer that is able to process such statements.
- Members of types are not yet completely resolved (e.g., if the type represents an array, accesses to elements of the array are ignored).
- When representing the inheritance between classes the visibility of the members is not considered, therefore all the elements are considered pubic. We still have problems with the correct recognition of overloaded of class functions and procedures.

- Our fact extractor only recognizes calls to libraries and third party software when we have access to the source code of this software. There might be situations in which an entity in the source code depends somehow upon some other entity defined in a library (e.g., inheritance). This leads to a lack of information. For this reason, it is also possible that some information (elements and relations) could be missing, unless the source code of such a library is available.

5 Related Work

Meta-models exist basically for any programming language. In this section, we compare our concrete meta-model for the Delphi programming language against the Dagstuhl middle model (DMM) as presented in [9]. The DMM is a programming language independent meta-model that aims at facilitating the interoperability of reengineering tools because of its independence towards specific programming languages. Both, our meta-model and the DMM represent the static structure of software systems. They have a wide overlap of similar elements (sometimes with different names but equivalent meaning), but there some differences, too:

- One strength of the DMM is that handles unpreprocessed source code, as well as visibility issues (i.e., private, protected, public declarations of meta-model entities), which we see as an open point of our meta-model.
- The DMM does not fully support the Delphi programming language, which is needed for our work. In detail, DMM supports neither namespaces (units in the Delphi context) nor properties. We consider these two elements as important concepts of the Delphi language, which are supported by our underlying meta-model. Therefore we miss as well the distinction between application units and third-party library or system units.
- Since the DMM is aimed to be language independent, it has to take care of elements and relations that are not needed for fact extraction from Delphi source code. The three hierarchies allow more details to be represented, but this brings an increased complexity into the meta-model. Our meta-model is targeted directly to Delphi and has to support only our needs so that we have sufficient information for our future reverse engineering tasks.

The different objectives of the DMM group and our work result, of course, in two different meta-models, but we see that the Delphi characteristics can be integrated into the DMM.

6 Next Steps

The first step in our future work is obviously to address the open issues as mentioned in Section 4. In order to have a working and stable fact extractor, our main focus will go in this direction. Then we can extend the meta-model and the fact extractor so that visibility issues are taken into account as well. Furthermore, we will think about other extractable relations.
In the future, we plan to establish a second output format next to the RSF format, namely the graphical exchange language (GXL) [5], an XML-based standard exchange format for graphs.
We will conduct further case studies to validate the abilities of our fact extractor, and since the fact extractor is based on the introduced meta-model, this will be checked as well. And it is already planned that we apply the fact extractor in architecture recovery projects. We will use the low-level information about the software system to facilitate the recovery and documentation of the software architecture.
To keep up with the time, it is very likely that we extend the fact extractor first towards Delphi 7, and later towards Delphi .NET. We expect the underlying meta-model to remain stable.

7 Conclusion

When reverse engineering a software system with (semi-) automated tools, it is important to have a common meta-model which underlies the tools. In this work we introduced a meta-model for the Delphi programming language and reflected our first experiences with a fact extractor currently under development by applying it to two open source projects.
We expect the meta-model and the fact extractor to help us in being successful in future reverse engineering and architecture recovery projects with software systems implemented in Delphi.

8 Acknowledgements

We are grateful to the German Federal Ministry of Education and Research for partially funding our work under EUREKA 2023/ITEA-ip00004 'from Concept to Application in system-Family Engineering (CAFE)'.

References

[1] Borland, URL: http://www.borland.com.

[2] Chikofsky, E.J., J.H. Cross, *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software, January 1990, pp. 13-17.

[3] Delphi, URL: http://www.borland.com/delphi/.

[4] Guo, G.Y., J.M. Atlee, R. Kazman, *A Software Architecture Reconstruction Method*, Proceedings of the 1st IFIP Working Conference on Software Architecture, pages 15-33, San Antonio, Texas, USA, February 1999.

[5] Graph eXchange Language (GXL), URL: http://www.gupro.de/GXL/.

[6] IEEE Std 1471-2000, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE-SA Standards Board, September 2000.

[7] Kazman, R., S.J. Carriere, *View Extraction and View Fusion in Architectural Understanding*, Proceedings of the Fifth International Conference on Software Reuse, 1998.

[8] Krikhaar, R. L., *Reverse Architecting for Complex Systems*, Proceedings of the International Conference on Software Maintenance, ICSM 1997.

[9] Lethbrigde, T. *The Dagstuhl Middle Model: An Overview*, in Proceedings of First international Workshop on Meta-models and Schemas for Reverse Engineering", ELSEVIER Electronic Notes in Theoretical Computer Science (ENTCS), 2003.

[10] Lischner, R. *Delphi in a Nutshell - A Desktop Quick Reference*, O'Reilly & Associates, 2000.

[11] Mayrhauser, A. von, J. Wang, Q. Li, *Experience with a Reverse Architecture Approach to Increase Understanding* IEEE International Conference on Software Maintenance (ICSM), 1999.

[12] Model Scene Editor (MSE), URL: http://sourceforge.net/projects/mse/.

[13] O'Brien, L., *Architecture Reconstruction to Support a Product Line Effort*, Software Engineering Institute, Technical Note CMU/SEI-2001-TN-015, July 2001.

[14] Tilley, S.R., K. Wong, M.-A.D. Storey, H.A. Mller, *Programmable reverse engineering*, International Journal of Software Engineering and Knowledge Engineering, pages 501-520, December 1994.

[15] SQL Parser (SQLP),
URL: http://sourceforge.net/projects/gasqlparser/.

[16] Unified Modeling Language (UML), URL: http://www.uml.org/.