Abstract

During maintenance activities programmers make pervasive use of domain knowledge about the code that they maintain. Unfortunately, most of the current program analysis techniques do not explicitly take into consideration knowledge about the domain of the analyzed program. In this paper we advocate the need to use the domain knowledge explicitly in reverse engineering in order to enhance the current analyses with logical information about the implemented domain and to enable new program analyses at the conceptual level. We outline a set of basic ingredients needed by domain knowledge driven program analyses and sketch our approach to interpret program parts from the point of view of the knowledge of the application domain that they implement by defining the intentional meaning of programs.

1 The need for domain knowledge in program analysis

Programming is a knowledge intensive activity. In order to write a program, programmers need to understand its business domain, have knowledge of design and different programming technologies. The reuse of already existent code requires that programmers know about already existent implementations of certain domain concepts. Once the programs are built, in order to maintain them programmers need to understand the existent code and thereby to (mentally) link it to knowledge about the domain that it implements. The knowledge used during different programming activities can be about the business domain of the entire program (e.g. banking), the programming technologies that are usually contained in standard libraries (e.g. XML, graphical widgets), the design and architecture (e.g. design patterns), computer science (e.g. operating systems), or general knowledge.

The state of the art automatic reverse engineering techniques make use of little knowledge (if at all) about the application domain of the piece of code under analysis. They do not usually care whether under analysis is a piece of code implementing the GUI, transfer of data, or the core of the business domain of the program. Furthermore, they usually do not care whether a part of the domain is implemented de-localized in different program modules or whether a module contains the implementation of non cohesive domain concepts.

We advocate the need to put the relation between programs and the domain knowledge that they implement in the center of program analysis.

The use of domain knowledge is an enabler for lifting the abstraction level of traditional code analyses and for defining new program analyses at a logical level. Many of the existent analyses can be refined (or even redefined) by using the relation between the domain knowledge and the code. For example, the mapping of domain knowledge on programs can be used to improve the concepts location and concepts assignment, describe the redundancy in the implementation of concepts, or enrich the current architectural analyses.

Besides enhancing existent analyses, the systematic use of domain knowledge enables completely new analysis possibilities such as: assess the measure in which the modularization of a program mirrors the conceptual decomposition of a domain, characterize the design quality with respect to the the (conceptual) extensibility of programs, characterize the consistency and quality of naming of program elements, or evaluate the faithfulness and conceptual coverage of domain specific APIs with respect to their modeled domain. We stress that beside these usage examples, the systematic interpretation of program fragments with respect to the domain knowledge that they implement can open new directions in reverse engineering.

2 Basic ingredients for domain knowledge driven program analysis

In order enable the domain knowledge driven program analyses we need to map the domain knowledge to program fragments that implement it. The mappings should be done in a rich and systematic manner. For this, we need the following ingredients (illustrated in Figure 1):

Program abstraction. We need an adequate program representation that leaves out unnecessary details but is rich enough to allow the desired analyses. Whenever the abstraction is too detailed, the analyses are hindered with too much information. When the abstraction is too high-granular then the results
of the analysis are inexact.

**Semantic domain.** We need an adequate representation of domain knowledge with respect to which to interpret the program. The semantic domain should be flexible enough to represent the concepts from a wide variety of domains in a faithful manner (with minimal encoding bias) and detailed enough to enable the desired analyses. The limits of the semantic domain determines the limits of the logical analyses that can be done on the program.

**Interpreting programs.** We need an unambiguous mapping of parts of the program to the chosen semantic domain. The mapping assigns meaning to the program and is the key ingredient for the implementation of more advanced domain knowledge driven analyses. In order to be possible to analyze big programs, we need to recover (or approximate) the intentional interpretation automatically.

### 3 Our approach: intentional meaning of programs

In the following we sketch our approach to define the intentional meaning of programs and that is an enabler for domain knowledge driven program analyses [1].

**Program are knowledge bases.** Programs can be regarded as knowledge bases whose content is the program identifiers and the knowledge representation language is a subset of the programming language. We abstract programs as graphs whose nodes are the program elements (e.g. classes, methods) and whose edges are program relations defined between the program elements.

**Domain ontologies as semantic domain.** We propose to use domain ontologies for representing and sharing the domain knowledge. Domain ontologies are conceptual models that contain a set of named concepts in a specialization / generalization hierarchy and relations among them. These concepts and relations should represent a consensual agreement on the domain by a category of domain experts. The knowledge contained in the ontologies determines the possible analyses that can be executed. In our experience, even simple ontologies containing only taxonomies of concepts (and no other relations) can be useful for many analyses.

Off-the-shelf ontologies are usually too general to allow the analysis of arbitrary programs. We proposed two ways to obtain ontologies appropriate for program analysis: we extracted domain ontology fragments that cover the domain of programming technologies (e.g. XML, GUI, databases) by analyzing different domain specific APIs that implement the same domain fragments⁠¹; and we developed a method to manually build ontologies based on the program under analysis by reverse knowledge engineering.

**Assigning intentional meaning to programs.** We define the intentional interpretation to be a function that maps program elements (e.g. classes, attributes) to concepts from domain ontologies that are intended to be implemented. The intentional interpretation assigns an intentional meaning to different program parts. The meaning is “intentional” since we do not say whether a program part really implements a domain concept (e.g. the class “CPerson” implements (correctly) the concept “person” – similar to verification) but only that the concept is intended to be implemented in the program part (e.g. the class “CPerson” intends to implement the concept “person”). We also make a distinction whether a concept is defined, internally represented, or only referenced by a program element.

In order to recover the reference of domain concepts in code we use similarities between the names of program identifiers and the relations among them induced by the program on the one hand, and the names of domain concepts and their relations defined in the ontology on the other hand.

### References


---

¹ [http://www4.in.tum.de/~ratiu/knowledge_repository.html](http://www4.in.tum.de/~ratiu/knowledge_repository.html)