Reverse Engineering Strategies for Software Migration (Tutorial)
Alex Quilici
ICSE ’95. Proceedings of the 17th International Conference on Software Engineering, Pages 333-336

Abstract

• Addresses question of whether or not reverse engineering of legacy systems is doomed to fail.

Introduction

• Reverse Engineering Definition: "The process of deriving abstract formal specifications from the source code of a legacy system, where these specifications can be used to forward engineer a new implementation of that system."

• Assumptions for this definition:
  ➢ Process of deriving abstract formal specifications is completely automatic
  ➢ Specifications are at a sufficient level of abstraction to allow the change
  ➢ Time and effort for reverse engineering is less than the time and effort for reengineering without reference to code

• Central question: is this possible in a cost- and time-effective way?

• Example: Reasoning System's REFINE system - used Augmented Abstract System Trees (ASTs)

• Current state of the art is far from the goal.

• Argues for revised definition: "The automated or assisted process of deriving a knowledge base describing a legacy system from its source code, where this knowledge base lessens the effort required to forward engineer a new implementation of that system." [my view: this definition is a cop-out]

What Can Be Automatically Extracted

• Extraction of partial specifications through various means.

• Extraction of higher-level abstractions at multiple levels using code pattern matching.

• Extraction of specifications via domain-specific pattern recognition.

• All techniques pose two more questions: can they be made complete, and can they scale?

The Completeness Question

• Pattern-matching is always doomed to incompleteness due to incompleteness of pattern library.
• Dependency on ratio of idiosyncratic to stereotypical code.
• Concludes domain-limited completeness may be within reach.

**The Tractability Question**

• Most existing algorithms are NP complete in the worst case.
• Successful program understanding must scale both with the size of the program and the size of the [pattern] library.
• Scaling problems inherent in addressing large programs may possibly be handled by modularization. [not likely in the general sense - modularization of traditional code will usually lose significant architectural information.]
• Cooperative Extraction
• This seems to be his main point - give up and be pragmatic to make it work
• Provide intelligent assistance to programmers who are attempting to understand the legacy system. [assuming lack of automatic understanding]
• Vision: understanding becomes a shared task between tools and programmers.
• Requires a single knowledge base for both human and machine information.
  ➢ Formally represented
  ➢ Visually presented
• Admits to other possibilities for approaches.

**Conclusions**

• "Old notion" of reverse engineering may be doomed to failure.