Chapter V

Specifying Coherent Refactoring of Software Artefacts with Distributed Graph Transformations

Paolo Bottoni, University of Rome “La Sapienza”, Italy
Francesco Parisi-Presicce, University of Rome “La Sapienza”, Italy
and George Mason University, USA
Gabriele Taentzer, Technical University of Berlin, Germany

Abstract

This chapter discusses the use of Graph Transformations for refactoring. Refactoring changes the internal structure of a software system, while preserving its behavior. Even though the input/output view of a system’s behavior does not change, refactoring can have several consequences for the computing process, as expressed for instance by the sequence of method
calls or by state changes of an object or an activity. Such modifications must be reflected in the system model, generally expressed through UML diagrams. We propose a formal approach, based on distributed graph transformation, to the coordinated evolution of code and model, as effect of refactorings. The approach can be integrated into existing refactoring tools. Due to its formal background, it makes it possible to reason about the behavior preservation of each specified refactoring.

**Introduction**

Software is subject to changes and a piece of software may need changes for several reasons. One such reason is the introduction of new requirements that cause the need for design changes. The introduction of a new requirement can be a consequence of either the iterative development process chosen for the project that constructs the system incrementally, or the fact that the requirement was overlooked in the initial specification and design of the system. As a simple example, consider an application developed around a single specific algorithm. If a new algorithm to perform the same calculations (graph layout, for example) becomes available, it may be useful to modify the application to add the option of using the new algorithm.

Object-oriented programming has made many changes easy to implement, often just by adding new classes, as opposed to more traditional approaches requiring many modifications. But adding classes may not be sufficient. Even in the simple example above, the application must evolve by means other than class addition. If the designer has not foreseen the possibility of alternatives for the algorithm, the class with the original algorithm would probably need to be “split” into algorithm-specific elements and general ones, the latter to be “moved” to a new class that will then provide the means to choose between the two algorithms, placed in separate components.

Another reason for wanting to modify an object-oriented program is to be able to reuse (part of) it. As an example, consider the case of two teams developing two class libraries independently. The two libraries may contain different classes implementing the same basic objects (windows, lists) or the same operations to manipulate them with different names. In order to integrate the libraries, it is best to remove these inconsistencies, by changing one library to use the basic classes or the operation names of the other one. Simple modifications such as the change of an operation name are not easy to implement, as they require searches for the procedures that can invoke them or for the other operations that they would override with the new name.
A Service Oriented Architecture for Coordination in Collaborative Environments
www.igi-global.com/chapter/service-oriented-architecture-coordination-collaborative/66918?camid=4v1a

The Information and the Strategy Business
www.igi-global.com/chapter/the-information-and-the-strategy-business/205082?camid=4v1a