Detecting Patterns of Poor Design Solutions Using Constraint Propagation

Ghizlane El-Boussaidi and Hafedh Mili
Laboratoire de Recherches en Technologies du Commerce Électronique (LATECE)
Faculté des Sciences, Université du Québec à Montréal
B.P. 8888, succursale Centre-Ville, Montréal (Québec) H3C 3P8, Canada
{el_boussaidi.ghizlane@courrier,hafedh.mili@}uqam.ca

Abstract. We are proposing an approach for applying design patterns that consists of recognizing occurrences of the modeling problem solved by the design pattern (problem pattern) in input models, which are then transformed according to the solution proposed by the design pattern (solution pattern). In this paper, we look at the issue of identifying instances of problem patterns in input models, and marking the appropriate entities so that the appropriate transformations can be applied. Model marking within the context of MDA is a notoriously difficult problem, in part because of the structural complexity of the patterns that we look for, and in part because of the required design knowledge- and expertise. Our representation of design problem patterns makes it relatively easy to express the pattern matching problem as a constraint satisfaction problem. In this paper, we present our representation of design problem patterns, show how matching such patterns can be expressed as a constraint satisfaction problem, and present an implementation using ILOG Jsolver, a commercial CSP solver.

Keywords: Marking models, constraint satisfaction problems, transformations, design patterns.

1 Introduction

Earlier transformational approaches described software development as a sequence of property-preserving transformations that are applied to a set of user requirements to produce functional software that satisfies a number of quality requirements [14]. Design choices made during the development process lead to different transformation chains. This idea was recently restored by the OMG’s model driven architecture [11]. The basic idea behind MDA is to separate the specification of a system from its implementation on a specific platform. MDA proposes to specify a system as a platform-independent model (PIM), to specify platform models—models describing platforms—and, to transform the PIM to a platform-specific model (PSM) according to the chosen platform. Roughly speaking, the transformation from a PIM to a PSM is guided by a specification that provides the mapping between their entities. Generally these mappings are specified using a combination of model type mapping—defined on a meta-model level—and model instance mapping which uses a marking process where—depending on the target platform—the architect marks entities of the PIM.
In practice, the transformations are not fully automated because the mapping between the source and target models is incomplete or non-deterministic [13] and, the marking process remains essentially a manual process. Consequently, a major challenge in MDA is: 1) to specify these mappings precisely to be able to encode them in systematic procedures, and 2) to support an automatic or semi-automatic marking process to guide these procedures when applied to other requirements. In this paper, we propose an approach for automatically marking models especially in the context of design patterns application. This approach consists of recognizing in input models instances of the design problem solved by a design pattern (problem pattern), which are then transformed according to the solution proposed by the design pattern (solution pattern). In this paper, we look at the issue of identifying instances of problem patterns in input models, and marking the appropriate entities so that the appropriate transformations can be applied. Our problem patterns representation enables to express the pattern matching problem as a constraint satisfaction problem.

The paper is structured as follows. Section 2 gives an overview of our approach to representing and applying design patterns. We describe the marking process in section 3. In sections 4 and 5 we show how the pattern matching problem is expressed as a constraint satisfaction problem. The implementation is presented in section 6 followed by a discussion in section 7 and the conclusion.

2 Overview of the Approach

Design patterns codify proven solutions to recurrent design problems. Design problems come in many shapes and forms, but a significant number of such problems can be expressed as poor solutions to modeling requirements. We represent such design patterns by triples (MP, MS, T) (see Fig. 1), where MP is a model describing the design problem solved by the pattern, MS is a model describing the solution proposed by the pattern, and T a transformation that transforms an input model exhibiting an instance of MP by replacing that instance with the corresponding solution. Both MP and MS are metamodels to the extent that their instances are models. We envision a catalog of such triples in a modeling workbench. Given an input model, a designer can try to detect potential instances of the problems solved by the patterns of the catalog in this model, i.e. matching the MP components of the patterns, to the input model. A successful match marks the entities of the input model by the roles they play in the model of the problem (MP), and suggests that the design pattern may be applicable to this model. The designer may then apply the transformation rules to the so marked model. The outcome is the input model where an instance of the design problem addressed by the design pattern has been replaced by the solution proposed by the pattern. In this paper we describe the matching and marking process.

Consider the example of a class hierarchy that represents a taxonomy of classes that implement the same behavior—same set of methods. A proverbial example of such a hierarchy is the class hierarchy representing the node types of an abstract syntax tree for some language X. Assume that the set of node types is stable—new constructs seldom get added to language X—but that new methods are regularly added to manipulate our abstract syntax trees in different ways. With the traditional <class,