Detecting Inconsistency in the Domain-Engineering

S

Abdelrahman Osman Elfaki
University of Tabuk, Saudi Arabia

Yucong Duan
Hainan University, China

INTRODUCTION

Recently, Software Product Line (SPL) is a major technical paradigm to develop software products that has been used by big companies. There are two main phases in SPL: the domain-engineering phase and the application-engineering phase. In Domain-engineering Phase, software assets are collected and organized in suitable model for configuration. Domain-engineering Phase is a continuous process. When there are new assets, they are added to the existing assets. Cumulative aggregation (for the software assets) may produce some errors. The grouping of assets may be made at different times and by different groups of people. In some cases, there is a parallel development process, i.e., several people add assets (to develop domain-engineering) at the same time. Moreover, Ajila and Kaba (2008) have proved that the domain-engineering process is a dynamic for some of reasons, such as changes in: technology, user requirements, or business rules. Another reason why domain-engineering is complicated is that the nature of a feature, i.e. software asset, (whether optional or mandatory) can differ from product to product (Buhne et al., 2004). As a fact, a successful software product which is generated from domain-engineering is highly dependent on the validity of it. Hence, validation is a significant process within the domain-engineering.

Recently, validation of SPL has been discussed as an important issue (Benavides et al., 2010; Heymans et al., 2011; Eisenecker et al., 2012). Mannion (2002) defines validation in SPL as a mechanism that is used to ensure that a SPL can produce at least one product that can satisfy the constraint dependency rules. Lan et al. (2006) define validation in variability as a mechanism to check if the configuration output satisfies corresponding variability constraints (in a specific domain) or not. In this article, we define validation in domain-engineering as a method used to ensure the detection of inconsistency in the domain-engineering's and provide explanations to the modeler so that inconsistency can be detected and eliminated.

Formalization and reasoning represent the lifecycle steps for the automated validation of SPL. Formalization means modeling SPL using the standard method, which allows some standard tools to reason the model. In this work, we have used First Order Logic (FOL) to formalize SPL and used Prolog (Wielemaker, 2007) as a reasoning tool.

In this article, inconsistency detection in domain engineering has been discussed. The initial versions of this work are published in Elfaki etl (2008) and Elfaki et al. (2009) where the formalization of our approach, i.e., our notations has been described in details.

In this article, we analyse the inconsistency problem and define three types of inconsistency. First Order Logic rules are developed in order to detect inconsistency in the domain-engineering process. The definition of the three types of inconsistency and the detection of inconsistency in the domain-engineering process are our contributions to the literature in respect of this operation. In the literature, only direct inconsistency is detected at the configuration stage (Hemakumar, 2008; White et al., 2009)

This article is organized as follow: Background is states in section 2. The inconsistency detection operation are described and discussed in section 3. Future directions are presented in section 4. Conclusion is discussed in section 5.

Software Product Lines: the SPL is defined as a software engineering methodology for creating a collection of software products from a repository of software assets. Meyer and Lopez (1995) define the SPL as a

DOI: 10.4018/978-1-4666-5888-2.ch697

set of products that share a common core technology and address a related set of market applications. In an SPL, all software assets are collected in one place, i.e., in a software assets repository. The specific software product is created from this repository using specific techniques. In an SPL, there are two main processes; domain engineering and application engineering. On the other hand, domain engineering is defined as the first process in an SPL is domain engineering, which represents the domain repository and is responsible for preparing domain assets using the variability model. Generally, the domain-engineering process replaces the traditional Software Development Life Cycle (SDLC) by focusing on the production of a family of software products that share common assets.

The main objectives of domain engineering are to (Pohl et al., 2005): provide definition for the variability and commonality in SPL, define the set of software products that can be generated from SPL, and provide reusable software assets.

Detecting Inconsistency in the Domain Engineering

In a SPL, domain engineering contains software assets. In our proposed approach, a software asset has only two possibilities: variation point or variant.

***Definition**: Let f_i denotes a feature, VP_i denotes a variation point, and V_i denotes a variant, where $i \in I+$.*

Inconsistency is a critical error; it can prevent the production of any software product that has an inconsistency relation between its features. Inconsistency is also known as a conditional dead feature (Hemakumar, 2008). Inconsistency is identified by (Batory et al., 2006) as a particular research challenge. Inconsistency occurs as a result of contradictions in constraint dependency rules. This type of error is very complicated because it can take different forms and can occur between groups of features or between individual features. Inconsistency in a FM describes relations between features that cannot be true at the same time, e.g. (f_1 requires f_2) and (f_2 excludes f_1), which means selection of f_1 must be followed by selection of f_2, but selection of f_2 prevents selection of f_1. Therefore, these relations cannot be true at the same time. A SPL can contain some other complicated forms of inconsistency. For instance, ((f_1 and f_2 and f_3) requires (f_4 and f_5)) and (f_1 excludes f_5). This example describes the existence of features f_1, f_2, and f_3 together which requires the existence of features f_4 and f_5. At the same time, feature f_1 excludes feature f_5. Thus, these relations could not be implemented at the same time.

Moreover, a SPL can contain complicated forms of inconsistency such as VP_1 excludes VP_2 and VP_1 require V_2, where VP_1, VP_2 are variation points and V_2 is a variant belongs to VP_2. Inconsistency is very complicated because it takes different forms. Inconsistency can occur between groups of features, individual features, or between a group features and an individual feature. In the following, we define three types of inconsistency and we develop logic rules (based on FOL) to detect all the defined types of inconsistency.

Forms of Inconsistency

We categorize inconsistency into three forms: direct inconsistency, indirect inconsistency and inconsistency related to a common feature. In the following, these forms of inconsistency are discussed and the rules that detect each form are illustrated.

Direct Inconsistency

In direct inconsistency, all features are of the same type: variation point or variant. The relation (f_1, f_2) requires (f_3, f_4, f_5) means the existence of both f_1 and f_2 requires the existence of f_3, f_4 and f_5 together. Direct inconsistency can be divided in four groups:

- **Many-to-Many Inconsistency:** Here, a set requires another set while the required set excludes the first one, e.g., ((f_1, f_2, f_3) requires (f_4, f_5, f_6) and ((f_4, f_5, f_6) excludes (f_1, f_2, f_3));
- **Many-to-One Inconsistency:** A set of features has a constraint dependency relation (require/exclude) with one feature while this feature has a contradictory relation to that set, e.g., ((f_1, f_2, f_3) requires f_4) and (f_4 excludes (f_1, f_2, f_3));
- **One-to-Many Inconsistency:** One feature has a constraint dependency relation (require/exclude) with a set of features while this set has a contradictory relation to that feature, e.g., (f_4