Chapter 13

Model–Driven Impact Analysis of Software Product Lines

Hyun Cho
University of Alabama, USA

Jeff Gray
University of Alabama, USA

Yuanfang Cai
Drexel University, USA

Sonny Wong
Drexel University, USA

Tao Xie
North Carolina State University, USA

ABSTRACT

Software assets, which are developed and maintained at various stages, have different abstraction levels. The structural mismatch of the abstraction levels makes it difficult for developers to understand the consequences of changes. Furthermore, assessing change impact is even more challenging in software product lines because core assets are interrelated to support domain and application engineering. Model-driven engineering helps software engineers in many ways by lifting the abstraction level of software development. The higher level of abstraction provided by models can serve as a backbone to analyze and design core assets and architectures for software product lines. This chapter introduces model-driven impact analysis that is based on the synergy of three separate techniques: (1) domain-specific modeling, (2) constraint-based analysis, and (3) software testing. The techniques are used to establish traceability relations between software artifacts, assess the tradeoff of design alternatives quantitatively, and conduct change impact analysis.

DOI: 10.4018/978-1-61692-874-2.ch013
INTRODUCTION

Changes are inevitable in software development and maintenance. Software adaptation and evolution represent changes that occur throughout the software lifecycle from conception to termination, such that change management influences both cost and quality (Lehman & Belady, 1985). Thus, impact analysis, which identifies the ripple effects of proposed software changes, is beneficial before developers make actual modification to a software asset. However, it is challenging for developers to analyze multiple candidate options for changes and make decisions that may have significant consequences (Arnold & Bohner, 1993; Bohner, 2002). Furthermore, it is difficult for developers to understand the consequences of changes across various software assets due to the structural mismatch of abstraction levels at different stages of the software lifecycle (De Lucia et al., 2008).

The challenges of software change are even more problematic for a software product line, which supports the derivation of a wide range of software products (members of a product family) through composing or modifying the core assets of its architecture. Developers can make changes to the problem domain and/or application domain of a software product line either to enhance the core architecture, impacting all the derived products, or to add more products as new members in the product family. Making changes to a software product requires the consideration of multiple constraints from different stakeholders and users of the product line family. It is possible that one stakeholder proposes a change to the requirements to maximize the value of his/her own product derived from the product family, but the change may positively or negatively influence other products or other properties of the product family.

Impact analysis (Arnold & Bohner, 1993; Bohner, 2002) accepts as input a root asset to which an initial proposed change is made, and then performs three main steps: (1) The analysis traces the relationships between the root asset and other assets to identify related assets; (2) The analysis examines each related asset to determine if it will be affected by the proposed change, and if so, what changes must be made by developers to accommodate the initial proposed change; (3) The analysis adds the effort to make additional changes on related assets to the total needed effort, producing an estimated scope and cost of the proposed change as the results of the analysis. Analyzing the impact of changes before performing actual modification to an asset has been recognized as an important task in the software development lifecycle (Arnold & Bohner, 1993; Bohner, 2002; Jönsson, 2007; Anquetil et al., 2008). The analysis results can serve as a preliminary input to planning project costs and predicting software system quality (Ajila, 1995). Despite its importance, the majority of support offered in current requirements and design tools provides only limited functionality. For example, given multiple alternatives to accommodate a change, quantitative and automated techniques are needed to assess the tradeoffs of each alternative, to balance the constraints from different perspectives, and to minimize the impact on existing products. Furthermore, it is difficult to assess change impact on heterogeneous software artifacts generated at different stages of the software development process. Manually creating traceability relations (the basis of impact analysis) is time-consuming, error-prone, and tedious. Although predicting change impact facilitates project planning and quality prediction, it is often omitted because of these preceding obstacles.

The Unified Modeling Language (UML) has been widely used for system analysis and design and a large number of UML diagrams have been developed to assist with lifecycle concerns. Some researchers have proposed impact analysis based on UML models to accomplish changes in the system while minimizing potential consequences, such as cost overrun and intermingled evolutions (Briand et al., 2006; Briand et al., 2002). However,
the UML is a general-purpose modeling language that tries to do many things for a broad range of uses. This nature of the UML can often limit the specificity of the abstraction and representation in which a system is modeled. An alternative to the UML has been realized in the adoption of domain-specific modeling languages (Gray et al., 2007), which allow the expression of a model in a form that is more natural (in terms of the abstractions and visualization of the model) manner to model an application and product line. This chapter presents our approach for using domain-specific modeling, in conjunction with impact analysis, to understand the effect of changes on a product line.

This contribution describes our approach for impact analysis based on model-driven engineering that targets software product lines, addressing the challenges listed in this introduction. The approach uses domain-specific modeling techniques to automate establishment of traceability relations, adopts constraint-based analysis techniques to quantitatively assess the tradeoffs of design alternatives, and employs a systematic testing framework to conduct change impact analysis.

BACKGROUND

This section presents the necessary background information needed to understand our approach, including the design and construction of domain-specific models, analyzing the impact of changes, and evaluating the alternatives of accommodating changes. Model-Driven Engineering (MDE) serves as a backbone of analysis and design activities in software product line development, which is a two-stage process including the stages of domain engineering (Bayer et al., 1999; Kang et al., 1990; Tracz, 1995; Czarnecki, 2006) and application engineering. Configuration management records and manages every change activity and configuration item. Impact analysis is performed in the process of change management, which controls the entire process of change (e.g., identification of changes, authorization and validation).

SOFTWARE PRODUCT LINES

According to the Software Engineering Institute, “A software product line is a group of software products sharing a common set of features that satisfy a well-defined set of market needs and that are developed from a set of common core assets for a specific application domain” (SEI-CMU, 2010). The notion of software product lines has received substantial attention since the 1990s and has proven itself in a large number of organizations (SEI-CMU, 2010; Weiss & Lai, 1999). Unlike single product development, a Software Product Line (SPL) is characterized by three essential phases: core asset development, product development, and management of technology and organization. In the phase of core asset development, core assets such as platform characteristics and components are developed or mined through domain analysis and then a series of products are produced on the basis of these core assets under the prescribed attached processes in the phase of product development. The management phase manages and supports both development activities and organization structure to maximize their performance.

Many organizations have realized the benefits of software product lines, such as reducing time to market and product-development costs, improving process predictability, enhancing product quality, achieving large-scale productivity gains, and increasing customer satisfaction (Clements & Northrop, 2001). For example, Hewlett Packard (Toft et al., 2000) reported that they shortened time-to-market three-fold, reduced typical defect density by a factor of 25, and reused code up to 70% across participating printer product lines. However, it is quite difficult to achieve similar levels of benefits as in the HP case. The main reasons are that few organizations could use the
same domain model as HP even though they may be in the same business. Furthermore, each organization has different levels of software development capabilities. Most of all, the way that product development is performed with software product lines is quite different from development of a single product. Thus, success in adoption and institutionalization of a software product line requires an organization to evolve its development process and methods, as well as the structure of the organization itself.

Unlike single-product development, the development of a product as an instance of a software product line follows a two-stage process: domain engineering and application engineering. The domain engineering stage focuses on developing core assets that contain variability and are reusable throughout a complete product line. Domain analysis is a key phase in domain engineering. During the domain analysis phase, engineers model the common and variable features as a feature model and specify the constraints among features to transform them into reusable core assets (Clements & Northrop, 2001). These assets are the design elements of a product line architecture and the basis of a product instantiation, with a proper variability management mechanism. After the engineers identify commonality and variability of the domain during domain analysis, the engineers design a product line architecture and its domain components in the domain design phase. The engineers should design a product line architecture to be platform-independent and to accommodate necessary variation management mechanisms (e.g., dynamic reconfiguration of features at runtime) (Gomaa & Hussein, 2004).

Application engineering instantiates a concrete product that is specific to customer requirements. The goal of application engineering is to produce the derivation of products as product line members through the selection, customization, integration, or transformation of the product specification and the appropriate assets developed in domain engineering. Application engineering begins with application analysis, whose goals are to determine the application-specific requirements and to analyze the impacts of selection from various platform-specific architectural design decisions (e.g., hardware resources, programming languages, binding time, and middleware). In the application design phase, engineers create platform-specific architecture and assets by selecting and customizing core assets developed in the domain engineering phase.

MODEL-DRIVEN ENGINEERING (MDE)

Model-Driven Engineering (MDE) has emerged as a promising paradigm in software engineering by emphasizing the use of models not just for documentation and communication purposes, but as first-class artifacts to be transformed into other work products (e.g., other models, source code, and test scripts) (Schmidt, 2006). Models may range from general-purpose modeling languages such as the Unified Modeling Language (UML) to domain-specific modeling languages (DSMLs), which assist domain experts in working within their own problem space without being concerned about technical details of the solution space (e.g., programming languages and middleware). DSMLs also provide an accessible way to communicate with stakeholders who are not familiar with the fast changing technologies.

The general approach to developing DSMLs consists of three different modeling layers: the model, metamodel, and meta-metamodel (Kurtev, Bézivin, Jouault, & Valduriez, 2006). Each model layer defines a representation structure and a global typing system that is used by the layer beneath it (i.e., each model conforms to its defining metalayer). For example, a meta-metamodel is a model that defines a metamodel (i.e., the metamodel conforms to the meta-metamodel definition). The top-level meta-metamodel has a special definition – it can be used to define itself such that no higher
layer is needed. The MOF (MetaObject Facility) (OMG MOF, 2006) is an example of a common meta-metamodel. For example, the metamodel of UML describes the constituents of all well-formed models such as Use case, Class, relationships, and properties. The UML metamodel definition is specified in MOF. A model represents an instance of a metamodel, and specifies an actual software system. A model must conform to its defining metamodel.

According to the report of practitioners who have adopted MDE for software product lines (SPLs) (Weiss & Lai, 1999; Pohjonen & Tolvanen, 2002), MDE can provide the role of the technological backbone for supporting product line development. DSMLs and specialized tools such as model transformation engines and code generators have been adopted and used based on the concepts available in specific domains. For example, domain analysis normally models commonality and variability with a feature model. MDE provides tools necessary to model and manage commonality and variability in a software product line. Figure 1 and Figure 2 describe a metamodel for a feature modeling language (Figure 1), along with a sample feature model that is defined in this language (Figure 2). The idea of a feature model was proposed by Kang et al. (1990) to represent all possible commonality and variability of an SPL in a single model using features and their relationships. Mandatory features are the common features across the product lines. Alternative and optional features are features that represent variations of the products to accommodate specific product requirements. Optional features are used if more than one feature is selectable for the
Figure 2. Feature diagrams for mobile media application

a) Feature model representing application capabilities

b) Feature model representing domain technologies

c) Feature model representing operating environments
products but alternative features allow selecting no more than one feature.

To model a system as a feature model, the metamodel should define three key concepts of feature modeling as mentioned earlier: Mandatory, Alternative, and Optional features. The feature metamodel also defines the relationships among these concepts, such as mandatory-to-alternative, which means that a parent feature is mandatory, but has alternative sub-features. Figure 2 shows a feature model for the MobileMedia application, which we use in a case study for applying our approach (described later in this chapter). Each rectangle represents features that are Mandatory, Alternative, or Optional. Each line shows the relationship between features through different line styles and arrow heads. For example, the relationship emanating into or out of a Mandatory feature does not have an arrow head. A line emanating from an Alternative feature uses a diamond arrow head and a line emanating from an Optional feature uses a normal arrow head. A solid line represents the relationship within a feature space, which is categorized into capability, domain, operation, implementation, and feature types (e.g., Mandatory, Alternative, and Optional). A dotted line represents the relationship among domain features. A mandatory-to-alternative relation is represented as a dotted line with no arrow head and an alternative-to-optional relation is represented as a dotted line with a diamond arrow head.

**CONFIGURATION MANAGEMENT & CHANGE MANAGEMENT**

Configuration and change management play the roles of the artifact management backbone to develop high quality software. Configuration management tracks changes of all the development artifacts such that the artifacts are not limited to source code. Change management tracks the status of development activities such as tasks, defect removals, and feature enhancement requests. IEEE standard 1042-1987 (IEEE 1042, 1987) describes four main activities in configuration management: configuration identification, configuration control, status accounting, and audit and review. The main tasks of configuration identification are identifying configuration items that comprise the structure of the artifacts and their properties, and making them unique and accessible. After configuration items are identified, configuration control manages the lifecycle of each item by creating baselines. Status accounting assumes the role of recording and reporting the status of configuration items. A configuration item’s completeness and consistency are validated through audit and review (IEEE 1042, 1987).

From the early conceptual phase through phases of completion to retirement, a software product is constantly changing. The success of these changes is determined by whether the modified software meets its requirements and the changes finish on time and within budget. Thus, if the changes are not managed properly, software development and maintenance suffer from poor software quality, unnecessary rework, and failed changes. The second layer describes key information of the change management process, which begins by submitting change requests. Before proceeding to actual artifact changes, a change manager analyzes the impact of configuration items from the changes. After impact analysis, the change control manager determines whether to approve or deny the change requests. For an authorized change, the actual change is implemented and validated.

**THE ORIGIN OF SOFTWARE CHANGE**

According to Pineheiro et al (1996), software changes come from either the social and business context of the system or from the improved understanding of the constraints as the maturity level
of software development is increased. Thus, it is very difficult to list all of the causes of changes. Researchers have characterized and categorized various types of changes with the expectation that such categorization can assist engineers in producing software with a lower defect density by preventing the occurrence of common undesirable types of change. In addition, the classification of changes may help to understand the nature of a change to minimize the crosscutting impacts. Lientz & Swanson (1984) distinguished the types of software maintenance as perfective, adaptive, and corrective. Buckley et al. (2005) developed a taxonomy for changes from a mechanisms perspective. Their approach focuses on the “how, what, when, and where” of an evolutionary change (i.e., any change request, such as requirement changes, bug fixes, and regulation changes that can occur throughout the software lifecycle). Chaplin et al. (2001) proposed 12 distinct categories on the purpose of maintenance. Based on related literature, the origin of a change can be classified according to the location of the change:

- **Internal change**: An internal change is a change whose origin comes from the software development lifecycle. As software development proceeds, engineers better understand the requirements, system, and development infrastructures (e.g., database, middleware and programming language). This enhanced understanding causes changes; for example, engineers restructure a design to use several design patterns to minimize dependency on concrete implementations (Gamma et al., 1994), or adopt an N-tier architecture to enforce reusability of the system. In addition, a defect is an important source of an internal change, which fixes the defect.

- **External change**: Unlike internal changes, the origin of an external change comes from outside a development organization (e.g., marketing and customer demand, new or revised regulations, and feature enhancement).

<table>
<thead>
<tr>
<th>Table 1. Examples of changes in SPL (based on McGregor, 2005)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>Anticipated Evolution</td>
</tr>
<tr>
<td>Unanticipated Evolution</td>
</tr>
</tbody>
</table>

Change management in a SPL can be different from that of a single product. Changes in a software product line are more predictive and manageable because most changes are based on technically anticipated and proactively researched requirements that are related to advances in the technologies and business. As described by McGregor (2005), the core assets of a software product line are designed and implemented with numerous variation management mechanisms based on technology and business forecasting. Thus, the variation mechanisms that accommodate evolution of a software product line can be anticipated. Table 1 classifies examples of change according to the origin of the change and the types of evolution.

**IMPACT ANALYSIS**

As software systems continue to become more complex, estimating expected changes is more difficult and often leads to inaccurate predictions. Lindvail & SanDahl (1998) empirically showed that developers predict the impact of change optimistically, which results in the underprediction of actual changes by a factor of 1.5 to 2.2. This result suggests that even experienced developers can predict the scope of changes with at most 50%
accuracy. This underprediction of actual changes results in serious problems in software development and management. Thus, systematic impact analysis is necessary to improve the accuracy of the estimation of changes when change requests are introduced to the software development and maintenance phases.

Impact analysis identifies potential consequences of a change request and can predict what artifacts need to be added, deleted, or modified to accommodate a change request. Impact analysis normally follows two steps: Identification and Estimation. The identification step identifies potential consequences from the change. Traceability relations between artifacts play a pivotal role in identifying the potential consequences because the links connect between different software artifacts such as requirements, design documentation, source code, and test cases. Traceability relations help with the task of impact analysis in three main ways. First, a traceability relation assists in verifying that a system meets its requirements by tracing from requirements to tests, and vice versa. Thus, a traceability relation helps to determine whether a specification is completely implemented and covered by tests. Second, a traceability relation helps to comprehend the system under analysis. When a system is developed top-down, a traceability relation helps a maintainer or tester in finding relevant documents such as design and requirements documents, and assists in understanding the core concept for estimating the scope and cost of system changes. Finally, a traceability relation helps to analyze the impact of changes by locating needed information to be updated and recorded when there are change requests. After the identification step, the estimation step assesses the impacts of possible changes and computes an estimate for the necessary modification, schedule, and cost based on the potential consequences.

Although performing impact analysis before doing an actual modification offers several benefits to software development and management, it is often done only when absolutely necessary. Such uncommon practice is because the current practice of impact analysis typically is performed manually, such that the increasing volume of heterogeneous artifacts and their interrelationships is very burdensome to manage. In some tool chains, such as IBM Rational tools (IBM Rational DOORS, 2009), impact analysis can be automated through integration, but it is very rare that an organization focus its complete tool chain on just one vendor. Furthermore, there is no sufficient method to describe the semantics of software change relationships and the traceability of those relationships.

**MODEL-DRIVEN IMPACT ANALYSIS FOR SPLS**

This section presents our approach for performing impact analysis based on model-driven engineering (MDE), addressing the problems discussed in the previous two sections. The process of impact analysis is initiated from the submission of change requests. There could be multiple alternatives to accommodate the proposed changes. Assessing these alternatives before implementation is challenging because of the lack of models to express and evaluate both existing products and envisioned changes. It is also difficult to establish traceability relations between assets, which are represented through different abstractions depending on a specific development stage. For example, requirements are normally expressed with natural language and subsequent design may be expressed with visual models. Across the lifecycle stages, there is often a difference in the granularity of the software representation and a lack of automation to support analysis and evaluation of design candidates. Our approach consists of three integrated parts: building domain-specific models to represent both existing products and proposed changes, assessing tradeoffs of each alternative (to accommodate the proposed changes) using a constraint-based design testing framework, and
Model-Driven Impact Analysis of Software Product Lines

Figure 3. Procedure for model-driven impact analysis

![Procedure for model-driven impact analysis](image)

applying the selected alternative to accommodate the changes. Figure 3 shows the procedure for using our approach.

Our approach starts with building domain-specific models for requirements, designs, implementations, and tests. Each model is linked to each other through a traceability relation to verify that the descendent model satisfies requirements or constraints of its precedent, as well as to analyze the impacts of changes. When a change request is introduced to one of the models, an engineer may query the traceability relations to obtain information about which model elements should be modified to create candidate models. Based on this information, several different change candidates are constructed and these candidates are passed on to the constraint-based impact analysis part to select a model that maintains a high independence in the level of design and design volatility. The procedures for impact analysis can be done automatically given that the candidate change models are provided and traceability relations are established across all the artifacts. After traceability relations are linked to every artifact, adding or modifying traceability relations can be done semi-automatically by interacting with engineers. Each of the key parts in Figure 3 is described in the subsequent subsections.

BUILDING DOMAIN-SPECIFIC MODELS

The ability to define new modeling languages for specific domains of interest is a key advantage of MDE over general-purpose modeling languages that are fixed on a specific notation (e.g., UML). Domain-specific models for MDE are built from three main steps:

1. **Analyze the domain.** To better understand the requirements and needs for a specific modeling language, engineers should conduct domain analysis to recognize the usage scenarios and concepts that need to be expressed in a new modeling language. In this step, engineers define the exact scope of the domain model and clarify the intended use of the modeling tool, often through experts who would be users of the modeling language.
2. **Define a metamodel for the domain.** After understanding the domain and the needs of a modeling language for that domain, engineers then define a metamodel formally. The metamodel itself is specified using the language of a meta-metamodel, such as MOF (OMG MOF, 2006).

3. **Build a domain-specific modeling tool.** Many meta-configurable modeling tools can generate a new modeling environment from a metamodel, such as GME (GME, 2010) and MetaEdit+ MetaEdit+, 2010). However, engineers must consider the integration of domain-specific modeling tools with the existing development environment before adopting a new modeling language.

As shown in Figure 4, at least six models (i.e., feature model, design model, implementation model, test model, impact analysis model, and trace model) are required to analyze the impacts of potential changes. However, the number of models that need to be considered during impact analysis may vary depending on the objective of the impact analysis. For example, if the primary interest is on the impact of test cases from potential changes, the only analysis that is needed involves the actual change request and the candidate test cases that are affected by the request.

The first step of our approach is to model all the variations of a software product line by using a DSML. Each modeling language has a corresponding metamodel that captures the essence of the domain. It may be possible to use several different general-purpose modeling tools and DSMLs. However, they should provide sufficient APIs to integrate with each other. Particularly, to support a software product line, the available tools may have capabilities to capture the common and variable properties that are required to analyze the impact of changes across the lifecycle (e.g., from the very beginning of the development to the deployment of the project) or versioning its model evolution.

The second step is to construct an infrastructure for the impact analysis (e.g., traceability relation) between each model. Although the traceability relation allows traversing bi-directionally, the traceability relation is actually intended to point artifacts from the subsequent development phase to its precedent. For instance, traceability relations between requirements and domain model elements...
are created so that each domain model element can point to its relevant requirements. Engineers must generate the initial traceability relation by indicating the source and destination of a relation manually from a tool called the traceability relation manager. After the initial traceability relations are established, engineers can manage the traceability links in a semi-automated way based on the metamodel designed for a specific software product line. If a change is introduced in one of the models, a traceability relation manager queries and gathers relevant information for this change to construct an infrastructure for the impact analysis (e.g., traceability relation). This procedure may be guided through the impacted artifacts by highlighting unmatched model elements that are missing traceability relations or have suspiciously matched model elements (i.e., the impact changes of model elements must be examined for those that have an indirect traceability relation).

The third step is to analyze impact and design tests based on constraints. This step is described in the next section and can be performed automatically by referring to traceability relations.

CONSTRANT-BASED IMPACT ANALYSIS & DESIGN TESTING

After establishing traceability relations within a domain-specific model, our approach further transforms the model into an augmented constraint network (ACN) that represents the assumptions and constraints from different perspectives embedded in heterogeneous software artifacts. A design testing framework (Cai et al., 2007) is similar to program testing, but considers the envisioned changes as test inputs. This testing framework uses ACN as a computational core and the test output is a set of quantitative modularity metric values, such as the volatility (Sethi et al., 2009) of the resulting design under given changes. To assess how each alternative impacts the modular structure of the product family, designers can test these alternatives under the same changes and compare these alternatives based on the computed metric values. In this section, we first introduce the concept of environmental parameters and design rules that form the basis of our approach. Then we introduce the ACN that formalizes these concepts and forms the computational core of the approach. Finally, we describe the design testing framework, as shown in Figure 5. In particular, we introduce the modularity metrics that are used as the output of our framework.

Environmental Parameters and Design Rules. Sullivan et al. (2001) introduced Environmental Parameters into software models to capture external factors that drive software changes. In a software product line, features are one kind of environmental factor because features drive changes to the product line and are usually not controlled by the designers. According to Baldwin & Clark (2000), a design rule is a stable decision that decouples otherwise coupled decisions and thus creates independent modules. A module is independent if it depends only on design rules, but not on other independent modules. For example, MobileMedia adopts the model-view-controller (MVC) architecture, in which the view (user interface) components and controller components use the Java Command API as the communication framework so that the UI components are decoupled from the controllers. Accordingly, the Java Command API is a design rule. Design rules are the most important architectural decisions that frame the modular structure of the design and influence its stability.

Given a software design, our design testing framework tests its modularity variation under a changing environment. To compare two design alternatives, we test them against the same set of environmental parameters to see which alternative generates the least impact on the original design, and which one produces the best modularized and most stable structure, determined by their design rules. Uniformly modeling environmental and design variables is the first step toward automated
design testing. Next, we introduce such a modeling technique.

Augmented Constraint Network. Cai (2006) developed the ACN as a logic-based design modeling technique, which consists of a set of variables that can model any dimension of software, including environmental conditions and design rules. A variable with a given value models a concrete decision or condition in a dimension. Their relations are modeled as logical constraints. An ACN also includes a dominance relation (DR) that formalizes the notion of a design rule - the asymmetric dependencies among design decisions, and a cluster set (CS) in which each cluster models one way a design can be aggregated. In the rest of the chapter, we refer to an external condition as a concern variable or environmental variable, and interchangeably refer to a design decision as a decision, design variable, or variable.

ACN entails a paradigm-agnostic means to compute modularity properties of a design, and precisely defines a pair-wise dependence relation (PWDR) among design decisions: if a decision y depends on x, that is, \((x, y) \in \text{PWDR}\), then there must exist a consistent state of the ACN. Changing the value of x violates some constraints and makes the constraint network inconsistent, and the value of y needs to be changed in a way that restores consistency.

From an ACN, a Design Structure Matrix (DSM) (Baldwin & Clark, 2000) can be automatically generated to visualize the modular structure of an application. A DSM is a square matrix in which the columns and rows are labeled with design variables, and a marked cell is used to model that the decision on the row depends on the decision on the column. Figure 4 depicts a DSM that models one release of MobileMedia. The blocks along the diagonal represent the modules in the system. All the DSMs shown in this chapter are automatically derived from ACNs. The first block includes seven variables and denotes the features in this release as environmental conditions. The second block contains the design rules of the architecture. The rest of the blocks represent the components. We uniformly represent environmental conditions, design rules, and other design decisions as an ACN, which we then solve to generate pair-wise dependency relations that capture the modular structure of the design. Next, we assess change impact with a design testing framework.
A Design Testing Framework. Figure 5 illustrates our design testing framework. The test input consists of two parts: a software design, which can be represented using prevailing design models (such as a UML class diagram, domain-specific model, or a component diagram) and the environment that this design is embedded, such as requirements and features. In the context of this chapter, a software design is a UML component diagram representing a software product line, and its environment is represented as a feature model. The output of the testing framework is a modularity property measured against a set of software modularity and stability metrics. Within the framework, we translate the design model into an ACN (UML2ACN) (Wong & Cai, 2009, Sethi et al., 2009) that contains only design decisions. Given the feature model, we establish a traceability relation, which we also translate into an ACN that represents the environmental parameters of the design (TL2ACN). We use a variable to represent each feature and use a constraint to model the assumption between a feature and the components that implement it. We integrate these two ACNs into one ACN (Integrated ACN), from which we generate a PWDR and calculate the modularity and stability metric values. In this section, we mainly introduce the stability and modularity metrics (Sethi et al., 2009) whose values are the output of our framework.

Design Stability Metrics. Software stability is usually measured based on how software components depend on each other syntactically, such as the number of classes outside a package that depend on classes within the package. However, it is possible that some part of the system is highly coupled, but is not subject to any environmental changes. As a result, the design could have a low stability value, but in reality, is highly stable. We thus use a DecisionVolatiliy metric to measure the stability of a design decision, \( x \), and a DesignVolatiliy metric to measure the stability of the whole design.

As introduced in our prior work (Sethi et al., 2009), the DecisionVolatility metric assesses the stability of a decision in terms of the number of environmental conditions that influence it (EnvImpact) and its own impact scope (ImpactScope). The rationale is that the more environmental conditions influencing \( x \), the more likely \( x \) will be subject to change; the more decisions \( x \) can influence, and the more impact \( x \) will have on the stability of the whole design. The DesignVolatility is thus the summation of all individual DecisionVolatility values. We chose these metrics because in model-driven systems, features and domain requirements are usually explicit and integrated with the underlying design and implementation models, and these stability metrics directly measure the impact of these environmental conditions, unlike other prevailing coupling and cohesion based metrics.

Figure 4 shows the MobileMedia DSM in which variables are clustered into environment, design rule, and component blocks (shown as blocks consisting of two variables). This DSM visually shows how the volatility metrics can be calculated from the PWDR relation; the numbers in the column to the right of the DSM are the total number of environmental variables that influence the variable on the corresponding row. The numbers in the row next to the last row of the DSM are the impact scope of the variable on the column. The numbers further below are the DecisionVolatility value of each variable, which sums to the DesignVolatility value shown in the cell with dark background and white text. From the DecisionVolatility values, we identify the most unstable variable.

In this case, AlbumData_Interface seems to be the most volatile decision (DecisionVolatility=21). Although it influences only three other variables, it turned out to be heavily influenced by multiple environmental conditions. This observation is confirmed by MobileMedia designers: whenever a feature is added, deleted or changed, the data set that needed to be accessed has to be changed,
Model-Driven Impact Analysis of Software Product Lines

hence a change in the AlbumData_Interface. It is important to note that, for the sake of simplicity, the Volatility metrics count only the number of environment variables that impact design decisions. They do not model how likely these environmental conditions will change. It is possible that a design variable suffers impact from many environmental conditions, but none of them will change. As a result, the variable may have a very high volatility value measured by the analysis but be highly stable in reality. The volatility metrics can be extended with the probability of change for each environmental condition. We can then use the extended metrics to conduct sensitivity analysis to assess design stability under uncertainty.

**Independence Level (IL) metric.** The key property of a modular structure is to allow tasks to be accomplished in parallel and independently. According to Baldwin & Clark (2000), a module generates value in the form of options: “a module creates an option to invest in a search for a superior replacement and to replace the currently selected module with the best alternative discovered, or to keep the current one if it is still the best choice.” Intuitively, the more independent modules there are in an architecture, the higher the option values that can be generated. Prevailing models, however, use classes, aspects, or components as modules, which are usually not independent. To explicitly measure how a system supports independent task assignments and option generation, we chose an Independence Level (IL) metric introduced in our recent work (Sethi et al. 2009, Wong et al. 2009) to measure design modularity.

The IL metric quantifies the extent to which a design can support module-wise independent searching and replacement; that is, its ability to generate option values. Baldwin and Clark’s net option valuation (NOV) statistically accounts for the options value embedded in a software structure, requiring the user to estimate a number of economic parameters based on software modular structure, such as the technical potential and cost of each module. We identify independent modules from an architecture as the first step towards more sophisticated option value reasoning.

To identify independent modules, we first cluster the variables in the PWDR generated from the ACN into a design rule hierarchy (DRH) using our previous algorithm (Wong et al., 2009). Figure 6 shows a DSM clustered into a DRH from the DSM shown in Figure 4. The DSM shows a hierarchy with three layers. The lower-right block (white background) contains independent modules that depend on the layers preceding them. The variables in cluster “Level1” contain decisions that only make assumptions about the decisions in cluster “Level0” and influence the decisions in the lower-right block. Moreover, once the decisions in Level0 are made, the clusters of decisions in Level1 can be made independently and concurrently. The only truly independent modules are the clusters in the lower-right block. All other variables are environmental parameters or design rules that make these modules independent from each other.

Clustering a DSM into a design rule hierarchy reveals several modularity properties. In Figure 6, the cells show that a number of dependencies are clustered. For example, there are 33 decisions in Level1 that depend on decisions in Level0. The DSM thus shows the impact scope of each variable and each level. In addition, because only the modules in the last level are independent, the more variables in the last level, the larger part of the

![Figure 6. A DSM that is clustered into a Design Rule Hierarchy](image)
system can be freely swapped or evolved under stable design rules, and the better the design is modularized. We thus define a simplified option-oriented Independence Level metric as the percentage of the variables falling into the last level of the design rule hierarchy. For example, the Independence Level of the MobileMedia design is 0.28, meaning that about 28% of the MobileMedia design is independent and can effectively generate option value.

Another stability metric that we consider is change impact. Similar to a conventional change impact metric that quantifies changes in code, this metric quantifies modified elements in an ACN model to analyze, for instance, the satisfaction of pivotal design principles, such as the open-closed principle (i.e., software is open to extension, but closed to modification) (Meyer, 2000). By comparing the number of variables that are added, removed, or changed in two ACNs that model consecutive design releases, we can easily calculate the number of variables that are added, removed, and changed.

ESTABLISHING TRACEABILITY RELATIONS

Impacts are analyzed based on traceability relations, which link heterogeneous artifacts that are developed in different lifecycle stages. The variety of artifacts makes it difficult to create the traceability relations correctly and consistently. Thus, one of the key parts of the impact analysis is to devise a method for establishing traceability relations to identify the relevant artifacts quickly and correctly. Various techniques have been proposed to create and maintain traceability relations, including cross referencing schemes (Evans, 1989), keyphrase dependencies (Jackson, 1991), requirement traceability matrices (Davis, 1990), matrix sequences (Brown, 1991), hypertext (Kaindl, 1991), integration documents (Lefering, 1993), assumption-based truth maintenance networks (Smithers et al., 1991), constraint networks (Bowen et al., 1990).

Some practitioners (Alves-Foss et al., 2002; Maletic et al., 2005) have adopted XML to represent traceability relations. As XML is platform-independent and vendor-independent, it is frequently used to exchange messages between heterogeneous systems; store, retrieve, and process documentation; manage information, and configure system environments. For example, most commercial and open source software design tools (e.g., IBM Rational Rose, IBM Rational Rhapsody, ArgoUML, SysML, and MagicDraw UML) adopted XML as the representation medium for storing design models and configuring tool options. In addition to XML, the World Wide Web Consortium proposed three language derivatives from XML: XLink (XLink, 2001), XPath (XPath, 1999), and XPointer (XPointer, 2002). These languages allow traceability relations to be embedded into models and other specifications (e.g., source code and test cases).

XLink is the XML link language that was designed to be used in many domains that need to link software applications. XLink defines a link through universal resource identifiers and it allows XML documents to assert linking relationships among more than two resources, associate metadata with a link, and express links that reside in a location separated from the linked resources. XPath is the XML Path language, which is a query language that uses navigation path expressions to select nodes or node-sets in an XML document. XPointer is the XML Pointer language, which is used as a basis for fragment identifiers for any resource whose type is encoded according to the XML standard. Because most models used in our approach adopt XML to represent the information needed for impact analysis across software artifacts, we use XLink and other XML language derivatives to specify the features needed for traceability relations. Listing 1 shows the part of an XML document type definition (DTD) that defines the traceability relations between a
**Model-Driven Impact Analysis of Software Product Lines**

**Listing 1. Example of a Traceability Relation Between a Feature Model and Class Diagram**

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<link-from-class-to-feature>
<!--Comment-->
<Files SourceFile="MobileMedia.dia"
      TargetFile="MobileMedia.fm"/>
<link desc="Link from AlbumListScreen to Create photo album"
      source_name="AlbumListScreen"
      source_path="/dia:diagram/dia:layer/dia:object[@type='UML - Class'][1]"
      target_name="Create photo album"
      target_path="/RealRoot/ModelContainer4Man"
      relation = "Refined"/>
<link desc="Link from AddPhotoToAlbum to Add photo"
      source_name="addPhotoToAlbum"
      source_path="/dia:diagram/dia:layer/dia:object[@type='UML - Class'][3]"
      target_name="Add photo"
      target_path="/RealRoot/ModelContainer4Man"
      relation = "Refined"/>
<link desc="Link from PhotoViewScreen to Delete photo album"
      source_name="PhotoViewScreen"
      source_path="/dia:diagram/dia:layer/dia:object[@type='UML - Class'][6]"
      target_name="View photo"
      target_path="/RealRoot/ModelContainer4Man"
      relation = "Refined"/>
</link-from-class-to-feature>
```

feature model and class diagram. Each relation is generated automatically by using the traceability relation manager. The types of links are described as XML tags in the form “link-from-<source>-to-<target>”. For example, Listing 1 describes link information from a class diagram to a feature model, such that the link type is specified as ““link-from-class-to-feature>” in the second line.

**CASE STUDY: MODEL-DRIVEN IMPACT ANALYSIS OF A MOBILE MEDIAL SPL**

This section introduces a case study that demonstrates the application of model-driven impact analysis to an example software product line in the domain of Mobile Media (Young, 2005), which represents multimedia software that supports several different mobile phones (e.g., Android, iPhone, and Blackberry). The products used in the case study demonstrate various combinational features according to a customer’s requirements and the device’s hardware constraints. In particular, the case study explores the use of these devices in a collaborative environment where new requirements and features become necessary. The product is thus subject to a number of heterogeneous changes, including enhancing the core architecture, or adding a new application to the product family. A key part of the case study is the demonstration of how various models are
used to analyze the impact of changes, and how to use the design testing framework to select and assess the optimal candidates.

MobileMedia is an evolved version of MobilePhoto that was developed at the University of British Columbia (Figuerido et al., 2008). It is designed to support different software product lines for various phone vendors (e.g., Apple, RIM, Motorola, and Nokia) and exploits both object-oriented and aspect-oriented techniques. MobileMedia can handle photo, video, and music data on mobile devices, such as cellular phones. Throughout its evolution, MobileMedia was released through eight different versions of design and implementation. Each version added some new functionality or restructured the previous version to achieve an improved modularized structure. In some versions, feature types were changed from mandatory to alternative.

Release 1 of MobileMedia was the core design that only included photo-related functions, such as adding or deleting photos and photo albums. Release 2 added exception handling as a mandatory feature. Release 3 added photo labeling as a mandatory feature and photo sorting as an optional feature. Releases 4, 5, and 6 each added an optional feature, such as specifying favorite photos. Release 7 involved changing the mandatory characteristic of photo manipulations to become an alternative feature, and adding another alternative feature, music manipulation. The final release added another alternative feature that provided

---

**Table 2. Capabilities of the MobileMedia Product Line. Adapted from (Young, 2005; Figuerideo et al., 2008; Huynh et al., 2008)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Feature</th>
<th>Brief description</th>
<th>Constraints</th>
<th>R1</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandatory</td>
<td>Create media folder</td>
<td>Create new media folder to categorize media to be added</td>
<td></td>
<td>P</td>
<td>M</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add media</td>
<td>Add photo, video, or music to the file system</td>
<td></td>
<td>P</td>
<td>M</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Delete media</td>
<td>Delete photo, video, or music from the file system permanently</td>
<td></td>
<td>P</td>
<td>M</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Label media</td>
<td>Label photo, video, or music with text for search purposes</td>
<td></td>
<td>P</td>
<td>M</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>View/Play media</td>
<td>Display a selected photo on the device screen; or, play video or music</td>
<td>Photo: View Video/Music: Play</td>
<td>P</td>
<td>M</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optional</td>
<td>Send media</td>
<td>Send media to others by transmission method</td>
<td>Predefined the transmission method</td>
<td>I</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specify favorite photos</td>
<td>Associate contact list with a photo in the device file system</td>
<td></td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sort media</td>
<td>Sort media by preference</td>
<td></td>
<td>P</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Keep multiple copies of photos</td>
<td></td>
<td></td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alternative</td>
<td>Caller Identification</td>
<td>Display photo for incoming call: Display caller’s photo if caller’s photo is linked with address book entry</td>
<td>Link between photo and address book</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Play melody for incoming call: Play customized ring tone per caller</td>
<td>Link between music and address book</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The evolution of MobileMedia features is summarized in Table 2. P, M, and V represent the media type of each release: photo, music, and video, respectively. Because the primary change request of Release 2 was focused on exception handling features, R2 is not shown in the above table.

In addition to Table 2, which shows the capability features of MobileMedia and the changes at each release, Table 3 through Table 5 capture additional domain information. Table 3 describes the media types and the related technologies that are key domain technologies for the MobileMedia system. Table 4 lists the operational environment of MobileMedia. Because MobileMedia executes on a mobile phone, it is largely affected by the mobile manufacturer’s development environments, such as a mobile phone library or the Mobile Information Device toolkit (Topley, 2002). Table 5 describes technologies that were used to implement MobileMedia. As MobileMedia was developed using Java and should support multitasking, the implementation of design patterns and thread management are important factors for implementation. However, Young (2005) classified features without a proper range of development decisions (i.e., the features were described only operationally in the sense of system capabilities). We found that this incomplete classification leads to missing traceability relations between feature models and class diagrams. For example, MobileMedia was designed and implemented to use a threading mechanism to avoid resource contention issues between device display and incoming phone calls.
However, the designed thread mechanism cannot be linked to its feature model because the original work considered only capability features. Thus, we employed a layered approach proposed by Young (2005) and recovered missing features through the traceability relations that we added.

**BUILDING DOMAIN-SPECIFIC MODELS FOR MOBILE MEDIA SPL**

To construct the foundation for model-driven impact analysis, we defined three different metamodels that represent languages for specifying feature models, traceability relations, and ACN. The metamodel for the feature model and an example instance model were shown in Figure 1 and Figure 2 in the previous Model-Driven Engineering section. For this case study, we used the Generic Eclipse Modeling System (GEMS) to define the various modeling languages and their instances (White & Schmidt, 2005).

Five entities and connections are defined for the metamodel for a traceability relation language. LinkModel is an entity that serves as a functioning container; this entity is not shown explicitly in a traceability relation instance. Both DesignModelSrc and DesignModelDst represent source and destination design elements for relations, respectively. The two models are linked by the relationship Refined. Unlike the refinement relationship in UML, the refinement relationship for a traceability relation is bi-directional. Thus, UML models can serve as the sources for a traceability relation with a feature model as the destination. In addition, other entities such as implementation, test cases, and test reports are defined to represent the key development artifacts within the impact analysis process. The relationships between entities are modeled through one of the named relations: Realized, Verified, Validated, and Tested. An instance of a traceability relation is shown in Listing 1. As discussed earlier, this figure illustrates the traceability relations between a feature model and a UML class diagram. Each traceability relation has a Refined relationship.

An additional metamodel is defined to represent a language for specifying ACN models. Such models can serve as an input model for impact analysis. The ACN metamodel can assist in guiding the transformation of a traceability relation into an instance representing an ACN. The ACN metamodel introduces inheritance to analyze the impact from the variations among features, design, or implementation. In an ACN, all model elements are generalized as a component with a body and an interface. Components are related to each other by a dominant relationship according to the vulnerability of the design or implementation changes. For example, interfaces dominate implementations. Normally, the decisions informing an interface change the way a component is implemented, but changes in an implementation may not force modifications to an interface.

Listing 2 shows part of an ACN instance that is based on the metamodel of ACN. This textual representation of the ACN instance corresponds to the input that is used by the ACN tool, and is generated from a model. The first three sections of the ACN representation are derived from an existing feature model or UML model. In an ACN, feature models are specified as variables and UML components are specified as two separate implementation and interface variables. The suffixes _impl and _interface represent an implementation body and an interface of a component, respectively. Each implementation or interface as a design dimension where decisions will be made, we abstractly model each dimension as having at least two possible decisions, one that is currently known and the other that is unknown, representing future changes. Accordingly, we model each variable as having two values, orig and other. In order to assess change impact, we care only that a decision will change, but do not care about what the decision is describing.

The fourth section models the constraint network for an ACN component. The constraints
Listing 2. An Example ACN Model (excerpt from MobileMedia ACN Model)

DesignSpace MobileMedia
{
  //Section 1:
  //ACN Component models which are derived from a UML Class diagram
  AddPhotoToAlbum_impl: {other,orig};
  AddPhotoToAlbum_interface: {other,orig};
  AlbumData_impl: {other,orig};
  AlbumData_interface: {other,orig};
  //Section 2:
  //ACN Component models which are derived from a feature model
  AlbumListScreen: {other,orig};
  AddPhotoToAlbum: {other,orig};
  PhotoListScreen: {other,orig};
  PhotoViewScreen: {other,orig};
  //Section 3:
  //ACN Component models for design rule
  ViewAlbumCommands_DesignRule: {other,orig};
  CreatePhotoCommands_DesignRule: {other,orig};
  DeletePhotoCommands_DesignRule: {other,orig};
  DeleteAlbumCommands_DesignRule: {other,orig};
  CreateAlbumCommands_DesignRule: {other,orig};
  //Section 4:
  //Constraint Network model derived from Class relationships
  AddPhotoToAlbum_impl = orig => AddPhotoToAlbum_interface = orig;
  AlbumData_impl = orig => ImageAccessor_interface = orig;
  ImageData_interface = orig => ViewAlbumFeature = orig;
  AlbumData_interface = orig => ViewAlbumFeature = orig;
  //Section 5:
  //Dominant relationship between classes
  [AddPhotoToAlbum_impl, AddPhotoToAlbum_interface];
  [AlbumData_impl, AlbumData_interface];
  [PhotoViewScreen_impl, Constants_interface];
  //Section 6:
  //Dominant relationship between feature and design rule
  [AlbumListScreen, ViewAlbumCommands_DesignRule];
  [AddPhotoToAlbum, CreatePhotoCommands_DesignRule];
  [PhotoListScreen, CreatePhotoCommands_DesignRule];
};
model the assumption relation between decisions. Instead of modeling the concrete states of each component and their relations, we abstractly model the fact that in any design, decisions are made for each component and other components make assumptions about these known decisions. For example, the first constraint in Section 4 models that the implementation decision on AddPhotoToAlbum is based on the assumption that AddPhotoToAlbum’s interface is as originally agreed. Finally, the last two sections specify the dominant relationship between ACN model elements, showing that the decisions on interfaces dominate implementation decisions and that the design rule decisions dominate other non-design rule decisions.

The metamodels representing traceability and ACN were not created to build modeling tools, but to assist in the transformation between the representations used to capture traceability and analysis information. For example, when a traceability relation and other models (e.g., a feature model or class diagram) are transformed, their representations are parsed by referring to their metamodels. The transformation assists in generating the ACN model, which conforms to its own metamodel.

**CONSIDERATION-LEVEL IMPACT ANALYSIS FOR MOBILE MEDIA EVOLUTION**

To illustrate how to perform impact analysis using the various modeling languages described in this section, assume that there have been two previous releases of the MobileMedia product line and a set of change requests have emerged for the next release (e.g., counting the number of times a photo has been viewed, sorting photos by viewing frequency, or editing the photo’s label). To analyze the impact of these change requests, a temporary feature model and a set of design models can be used to define a traceability relation between the anticipated changes. Two possible candidate designs are illustrated in Figure 7, which are refined from a temporary feature model. Both candidate designs have two common design decisions: (1) modifying the PhotoListScreen class to provide a sorted view that is based on the number of times a photo has been viewed and (2) introducing the NewLabelScreen class for editing the photo label. Each candidate design represents different approaches for managing the photo count and editing the photo label.

In design (a) of Figure 7, PhotoViewScreen has extended its functionalities as a controller to address new counting and editing of photo information. PhotoViewScreen manages the new changes by controlling the collaborations of two new classes, CountPhotoView and NewLabelScreen. In addition, PhotoListScreen references the view count information in CountPhotoView to display photos by its view preference. In design (b), the ImageUtil class was modified to manage the number of photo views by extending an attribute of each photo. Also, PhotoViewScreen was modified to count the number of photo views and pass the count information to the ImageUtil class. For editing the photo label, two new classes (PhotoController and NewLabelScreen) were designed.

When candidate designs are stable, the impact analysis of the new designs can commence by generating an ACN model using two tools (called UML2ACN and TL2ACN), which generate an ACN model from either a UML class diagram or a traceability relation model, respectively. The usage of such tools can be observed in the impact analysis process shown in Figure 5. The generated ACN model is then transformed into a DSM.

The DSM for this case study, as shown in Figure 8, provides information about the stability and modularity of the design, as well as illustrating the graphical design dependency. The Overall Design Volatility and the Decision Volatility measure the stability of the design, and the Independence Level for Design measures the modularity.
Figure 7. Candidate Designs for Release 3

a) Candidate design based on collaboration controller

b) Candidate design based on attribute extension
The Overall Design Volatility metric measures the stability of the whole design. The Overall Design Volatility is a summary of the Decision Volatility across the design, which measures the stability of a design decision. The Independence Level for Design computes the number of variables in an independent cluster relative to the total number of variables. Thus, a candidate design should be designed to minimize the Overall Design Volatility and maximize the Independence Level for Design. We also compute the change impact analysis in terms of the number of components added, deleted, and removed from the original design.

Table 6 shows that design candidate (a) is less volatile to change requests than design (b). In addition, although these two alternatives require adding the same number of new components and do not need to delete any component, design candidate (a) will modify fewer components, and better conform to the open-closed principle (Meyer, 2000). Although design candidate (b) has a slightly higher level of independence, the difference is not significant. The tradeoff shows that design (b) has only 1% higher level of flexibility but is 17% less stable. We conclude that design candidate (a) is a better choice.

To assess the proposed approach, we performed a systematic assessment to see if designers can confidently make decisions, such as which modularization technique is better in which circumstances without implementing the systems (Sethi et al., 2009). We applied the metrics to a series of releases of a software product line for MobileMedia. Both aspect-oriented (AO) and object-oriented (OO) editions of MobileMedia
were used and compared. Modularity and stability were key requirements in both editions, which underwent changes through 8 releases. UML component diagrams were available for all the AO and OO releases. Key external factors, the features driving new releases, were documented and considered. Their stability and modularity properties were computed based on the automatically generated ACN models and the proposed metrics.

Finally, we compared the findings obtained from our assessment with the conclusions obtained from in-depth analyses previously made from the source code (Figuerido et al., 2008). Our purpose was to compare the conclusions leading to decisions. The results showed that our approach reached highly consistent conclusions against that of implementation-level analysis. Moreover, the small numbers of discrepancies reveal issues in previously conducted implementation-level analysis. For example, our approach automatically detected several indirect dependencies between external concerns and internal components that were not picked up by source code analysis. These new metrics also led to new insights, e.g., showing how the superior option-generation ability observed in AO decomposition is paired with its lower stability. These positive results imply the possibility of faithfully assessing software stability and modularity at the level of higher-level models without the cost of implementation.

### Table 6. Summary of Impact Analysis for Candidate Designs

<table>
<thead>
<tr>
<th></th>
<th>Design (a)</th>
<th>Design (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall Design Volatility</td>
<td>103</td>
<td>121</td>
</tr>
<tr>
<td>Independence Level for Design</td>
<td>0.51</td>
<td>0.52</td>
</tr>
<tr>
<td>Components added</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Components modified</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Components deleted</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**FUTURE RESEARCH DIRECTIONS**

There are several remaining challenges and limitations that suggest areas for future work. The metrics that we presented in this chapter are focused on design issues. It would be a useful goal to increase the connection between design metrics and source code metrics as an effort to improve the transition between lifecycle phases as related to analysis of predictive change. Combining the results from a suite of metrics across the lifecycle may improve the ability to predict the overall cost of change and its effect on the project schedule. Many new areas of research can be investigated in this area of combining lifecycle metrics. An additional area for future work that we will soon explore is the extension of the functionality of the traceability relation to provide more query-specific opportunities (e.g., to enable the designer to issue queries based on suspicions that their own intuition suggests about a particular design and a set of change requests). Such query-specific traceability relations could assist in finding the depth of a link or a related attribute that appears in a trace.

**CONCLUSION**

This chapter introduced our approach for analyzing the impact of changes to software product lines (SPLs) through model-driven engineering. Software changes are a natural and inevitable part of a product line’s lifecycle, similar to single product development. However, impact analysis for SPLs is more complex and difficult than that of a single product development because software changes for SPLs can occur at any level of the system domain (e.g., either problem domain or application domain), and the impacts of these changes are propagated to the other domain. Thus, the support of automated tools and systematic assessment of candidate changes is highly necessary to produce impact analysis results that can be used to inform...
important decisions about software evolution. Our approach offers several benefits for domain analysis by exploiting domain-specific modeling languages and a systematic design testing framework for impact analysis. The domain-specific modeling languages help to identify and specify a domain’s characteristics, which consist of a number of different constraint sets, with graphical notations and make it possible to lift the abstraction and representation of a software change such that it can be analyzed and separated from specific platform and environment details that would exist at the implementation level. Traceability relations make it possible to detect and query multiple options for changes and multiple constraints from different stakeholders and users of the product line. By utilizing XML, a traceability relation can have a foundation to resolve the structural abstraction mismatch at different stages of the software development lifecycle. Thus, traceability relations can provide a streamlined foundation for domain analysis and it can reduce the effort to find relevant information across the domains for changes. In addition, a traceability relation provides information to generate constraint information, environmental parameters, and design rules, automatically for impact analysis. The results of constraint-based impact analysis provide guidance on which change option has significant consequences by analyzing multiple candidate options.

ACKNOWLEDGMENT

This work is supported in part by NSF CAREER award CCF-1052616 and CCF-1052616, and NSF grants CCF-0916891 and DUE-0837665.

REFERENCES


Model-Driven Impact Analysis of Software Product Lines


**KEY TERMS AND DEFINITIONS**

**Model-Driven Engineering**: Model-Driven Engineering (MDE) emphasizes the use of models not just for documentation and communication purposes, but as first-class artifacts to be transformed into other work products (e.g., other models, source code, and test scripts).

**Impact Analysis**: Impact analysis analyzes the range of impact from changes to a software system. The result of impact analysis is used to estimate the cost of such changes.

**Traceability Relation**: Traceability relation is the basis of impact analysis and informs how development artifacts are related to each other.

**Software Product Lines**: Software product lines represent a paradigm to develop a series of family products by maximizing the reuse of the commonalities among the software products and customizing variants for specific customers.

**Augmented Constraint Network (ACN)**: Augmented Constraint Network (ACN) is a way to measure modularity of the design, especially measure design volatility under given changes. The metric is formed from the Environmental Parameter and Design Rule.

**Environment Parameter**: An environment parameter represents factors that drive software changes. For example, changes in interfaces must drive changes in internal design and implementation.

**Design Rule**: A design rule is another factor that is used to make a software change. Unlike environment parameters, design rules are used to measure the dependency in internal design.