Developing configurable extensible code generators for model-driven development approach

Souvik Barat  
Tata Consultancy Services  
54-B, Industrial Estate  
Hadapsar, Pune, INDIA  
+91 20 66086261  
souvik.barat@tcs.com

Vinay Kulkarni  
Tata Consultancy Services  
54-B, Industrial Estate  
Hadapsar, Pune, INDIA  
+91 20 66086301  
vinay.vkulkarni@tcs.com

ABSTRACT
The need for agility and adaptiveness of business applications is on the rise with increased change in business dynamics. Code-centric techniques show unacceptable responsiveness in this dynamic context as business applications are subjected to changes along multiple dimensions that evolve independently. Use of model driven techniques for developing such business applications is argued as a preferable option as platform independent specification can be retargeted to technology platform of choice through a code generation process. Recent literature suggests the use of product line architectures to increase agility and adaptability by capturing commonality and variability, and configuring them appropriately. By visualizing a business application along five dimensions, namely, Functionality (F), Business process (P), Design decisions (D), Architecture (A) and Technology platform (T), the use of models are largely limited to F and P dimensions in commonly used model-driven development techniques and thus use of product line concept is also limited to these two dimensions. In this paper we argue the use of product line concept, i.e. variability and configurability, in these two dimensions is not sufficient to achieve the desired agility and adaptability. It is as well critical to use product line concept for code generators that addresses D, A and T dimensions. This paper evaluates existing approaches for code generator product line, introduces a new approach, and discusses early experiences with the presented approach.

Keywords
Configurability, Extensibility, Adaptiveness, Model-driven development, Code Generation product line

1. Introduction
Rapid evolutions of technology platforms and business demands have contributed to significant increase in business dynamics in recent years. The increased dynamics puts new requirement on businesses while opening up new opportunities that need to be addressed in an ever-shrinking time window. Stability and robustness seem to be giving way to agility and adaptiveness. This calls for a whole new perspective for designing (and implementing) IT systems so as to impart these critical properties.

Traditional business applications typically end up hard-coding the operating context in their implementation. As a result, adaptation to a change in its operating environment leads to opening up of application implementation resulting in unacceptable responsiveness.

Typical database-intensive enterprise applications are realized conforming to distributed architecture paradigm that requires diverse set of technology platforms to implement. Such applications can be visualized along five dimensions, namely, Functionality (F), Business process (P), Design decisions (D), Architecture (A) and Technology platform (T). A purpose-specific implementation makes a set of choices along these dimensions, and encodes these choices within application implementation in a scattered and tangled manner [13]. This scattering and tangling is the principal obstacle in agile adaptation of existing implementation for the desired change. Large size of an enterprise application further exacerbates this problem. This is an expensive and error prone process demanding large teams with broad-ranging expertise in business domain, architecture and technology platforms. Model-driven development alleviates this problem to an extent by automatically deriving an implementation from its high-level specification using set of code generators.

In last decade we have developed several business-critical enterprise solutions for a variety of business verticals like banking, financial services and insurance [13]. In our experience, no two business applications are exactly alike. For identical business intent, different enterprises, even from the same business domain, may have different requirements along these five dimensions with a significant overlap. Being ignorant of these similarities and differences would mean rework, and result in specification and implementation redundancy which will create maintenance and evolution problems later. For example, the functionality of a banking application can vary if it is targeting different geographies, e.g. USA and India. Similarly the same application can be targeted for delivery into different architecture and technology platform based on specific business requirements. Thus, it is important to capture commonality while highlighting the variability, i.e. product line architecture [5], to avoid multiple copies of same application with different choices of F, P, D, A and T dimensions. We are working towards a framework for developing adaptive applications such that variations (changes) along any of these dimensions can be incorporated easily within the desired time window. Our definition of adaptiveness subsumes configurability (i.e. selecting one of the many available variants) and extensibility (i.e. adding a new variant). We use model driven development approach for developing business application wherein a business application is specified in terms of platform independent models (conforming to a meta model) and high level specifications; and transformed into a deployable implementation using code generators. Each code generator translates a specific view i.e. generating a specific architectural layer e.g. presentation layer, business process layer, data access layer etc. We extend the architectural layer specific meta models to support family concept [4] through modeling of commonality and variability.
feature model [6] as a mechanism to select the desired member from the family i.e. the desired variant. Thus, we are able to model F and P dimensions of a business application product line [12]. We support extensibility along F and P dimensions by introducing placeholder model elements in the meta model which can be plugged in later with suitable models as new extensions. We have built upon aspect oriented techniques and applied them to models in order to realize our code generator product line [14]. Having tested this approach in practice with varying degree of success we observe this to be an effective approach for supporting business application product line (i.e. developing multiple applications with similar intent for a specific business domain for multiple situations i.e. users). However, we have identified some opportunities for improvement in the current implementation. In this paper, we present another approach to implement code generator product line on the lines of adaptiveness defined here.

The rest of the paper is organized as follows: we present a brief overview of developing adaptive applications using model driven development approach in section 2. Meta model for developing configurable and extensible code generator is presented in section 3. Section 4 presents an example. We discuss some of the related work in section 5. We conclude with a brief summary of early use of the proposed approach in section 6.

2. Model driven development of adaptive applications

In model-driven development approach, the various architectural layers of a typical business application are specified in terms of a set of models – each being an instance of a projection or a view of a unified meta model [13]. A, say, architectural layer specific code generator transforms the layer-specific model into the desired implementation on a chosen technology platform e.g. graphical user interface layer in JSP. Models, being at a higher level of abstraction, are easier to specify and are amenable for analysis leading to early detection of errors and prevention of some errors. Also, models can be kept independent of implementation technology platforms with model-based code generators that incorporate proven design and architectural patterns, deliver increased productivity, uniformly high quality, and retargetability.

Such specification-driven approach imparts adaptiveness to a certain extent as concerns related to D, A and T dimensions are separated from those related to F and P dimensions. For example, the same application specification, i.e. business functionality and business process concerns, can be delivered into multiple implementations that differ in terms of design decisions, architecture and technology platform of choices using different code generators. Thus adaptation related to D, A and T dimensions completely rely on adaptiveness of code generation process.

Fig 1 describes the process for deriving a purpose specific application from an application family specification. A deployable application for deployment choice \(<D_1, A_1, T_1>\) can be derived from application family specification by configuring application family to the desired application model (i.e. specification for application 1) and then transforming it into a deployable implementation using the appropriate code generator (i.e. code generator 1). The same application can be retargeted to a different deployment choice (i.e. \(<D_2, A_2, T_2>\)) by using different code generator (i.e. code generator 2). Thus, veracity of technology platforms leads to development of multiple code generators even for the same application specification. A code generator product line [14] addresses this problem as follows:

a) A code generator is a set of building blocks where each building block is a model to text transformer,

b) Model to text transformation process generates fragments of code corresponding to application models, and also the weaving specification to compose the fragments, and finally

c) Weave the generated code fragment as per the weaving specification.

We found, using this technique in MDD code generation raises several issues. For example - a) Lack of traceability from aspect to implementation affects readability, debugging and testing of the generated code as aspects are woven together in byte code, b) other issues related to (a), e.g. decomposition of system test cases and aspect level testing, computation of change impact of an application level change in aspect and c) tooling issues for using weaving tools for industrial-strength applications [14].
3. Extensible and configurable MDD code generator

A product line groups together a set of products (or product variants) which share a set of features (i.e. common part) and have distinguishing features (i.e. variations) [4,6]. Producing a purpose specific product can be seen as selecting from the available variants at each variation point. We model a family of code generators with a set of core and variable features. We capture a) feature commonalities with placeholders for extensions and variations, b) variability - a specific pattern for variations that can be plugged in at appropriate variation points, and c) extensibilities – a pattern for extending features as new variants of a code generator (code generator family). Fig. 2 shows the proposed code generator product line meta model describing commonality, variability and extensibility patterns for our code generator family. A brief description of the meta model follows:

- A code generator is composed of several composable transformational units. Each transformation unit encodes model-to-text (MTT) transformation rules.
- Transformational unit can be of two types – concrete and configurable.
- Concrete transformational unit encodes simple MTT rules without any placeholder for extension and variation.
- Configurable transformational unit is essentially MTT rules with placeholder for extension and variation. Meta model is extended to specify the extensions and variations as follows:
  o Body of a configurable transformation unit is specified using Template.
  o Each template can have multiple variation points (placeholder for variations).
  o A set of variants, which are realized as transformational unit, can fit into each variation point.
  o Having at least one variant for each variation point is necessary for defining well-formed transformational unit.
  o The meta model elements VariationPointType and RuleName (associated with parameters) is introduced to verify the semantic and syntactic correctness of the fitment between VariationPoint and Variant.

- A code generator product line can declaratively specified in terms of a set of features where each feature is associated with a transformational unit through realizedIn association.
- Configuration operation is equivalent to selecting one amongst the available variants for each variation points, i.e. selecting features conforming to FeatureRelations. FeatureRelation describes all possible feature relations [6], i.e. exclusion, inclusion, optionality, dependency, etc. The selected set of features, also called as feature configuration, must be internally consistent – whatever the definition of consistency may.
- Extension means adding a new variant for an existing variation points (i.e. adding variant of an existing feature) or adding a new Template by extending a transformational unit (i.e. adding new feature).

Essentially a code generator product line is a composite template whose behavior (i.e. transformation capability) is determined by a specific configuration which is conforming to feature relationships. We use an imperative model-to-text transformation specification language (OMGen [11]) for specifying transformation rules of a transformational unit as in our experience programmers are more comfortable with imperative languages than declarative languages. However declaration language, such as [8], can be used as specification language of transformational units. OMGen language provides for model navigation, model manipulation and text writing through familiar constructs such as IF, FOR EACH, PRINT etc. It also provides a syntactical short cut for PRINT construct through ‘.’ operator. The ‘.’ Operator is used for specifying model navigations and ‘$’ to distinguish model elements from normal text. It supports the notion of MODULE to organize transformation specifications into a set of reusable units called FUNCTIONS. We use Functions for implementing transformational unit, and extend OMGen language with a special keyword VARIATIONPOINT to denote variation point for a Function.

Earlier AOP based approach for realizing code generator product line used declarative model to text transformation specification
language, and was, in essence, a composition mechanism that relied on OO techniques like overriding to implement variability and inheritance to implement extensibility. The new approach uses an imperative OO language to specify model to text transformations and special syntax to denote placeholders i.e. variation points and extension points. This is identical to specifying variability and extensibility in business functionality and business processes. Imperative nature of specification language, and uniformity of mechanism for specifying variability and extensibility along all dimensions is the salient feature of the new approach.

4. Example

In this section, we illustrate our approach using an example - a simple code generator for generating java code from class model. Fig 3 shows a class diagram of a single class, Account, and desired code for two deployment scenarios, i.e. deployment environment 1 and deployment environment 2. As shown in the figure, the desired code patterns are similar for both scenario, i.e. an implementation of java classes with package statement, import statement, class header, modeled attribute, implementation specific additional attributes, constructor, getter and setter, method signatures with placeholder for method bodies. However, different deployment environments desire a little different code, e.g. environment 1 desires primitive data types for each modeled attributes (of primitive data types), and a specific set of implementation specific attributes (i.e. createDate, typeId, attributeSetIndicator), whereas environment 2 desires java objects for all modeled attributes and different set of implementation specific attributes. In addition, environment 2 desires code for logging method invocation information for each method. As requirements are different for these two environments, different code generator could be a possible solution for supporting these environments. However one can think of parameterized code generator with all possible options encoded within code generator implementation. But any extensions to such parameterized code generator might result into opening up entire code generator implementation and configuration could be unfeasible unless they are decided at design time. This limits the extensibility and configurability of a code generator to a large extent. We specify a code generator family as an instance of proposed meta model. Fig 4 depicts the model of a code generator family that supports code generation for environment 1 and environment 2 of the example described in Fig.3. As shown in the figure, code generator family, Class2Java, is composed of a configurable transformation unit, class2JavaGen. Class2JavaGen is further composed of concrete transformation units - printSuperClasses, importStatementGen, packageStatementGen. Essentially, these transformation units form the core (fixed part) of Class2Java code generator. Class2JavaGen has four variation points - printAttributeType, printInputParameters, printImplementationSpecificAttribute and printLogInformation to specify desired variations. The set of transformational units, shown in transformational units 2 of figure 4, are the available variants of supported variation points. For example – printAttributePrimitiveType and printAttributeObjectType are possible variants of printAttributeType variation point. Model describes the supported features of Class2Java code generators and feature relationships. The feature model described in Fig 4 specifies the following relationships - a) primitiveType and objectType are the alternate feature of AttributeType, similarly implementation choice 1 and implementation choice 2 are the available options of ImplementationOptions feature. b) MethodLevelLogging is an optional feature of Class2Java code generator, c) selection of implementation choice 1 is mandatory if primitiveType of AttributeType is selected. As shown in the figure, model describes the associations between supported feature and transformational unit that encodes the feature using
realizedIn associations. The OMGen specification of class2JavaGen is depicted in Fig 5. Fixed composedOf associations are encoded as IMPORT MODULE and EXTERN FUNCTION. The extended keyword VARIATIONPOINT is used for specifying all variation points of the function (Class2JavaGen).

A model of code generator family and organized transformation rules (encapsulated within transformational unit) improve extensibility and configurability capabilities into manifolds. For example, a specific configuration can serve code generator for a purpose specific solution; e.g. a configuration selecting PrimitiveType and Implementation Choice 1 of Class2Java code generator can produce the desired code for environment 1, and configuration selecting ObjectType, MethodLevelLogging and Implementation Choice 2 features can produce desired code for environment 2. Similarly the code generator model can be extended by i) introducing new variants of an existing variation point by adding new transformational unit – a mixed data type for modeled attribute, ii) replacing fixed composedOf association by variation point and providing possible variants – defining importStatementGen as variation point and providing variants, and/or iii) associating new template with different set of variation points for a transformational unit – a new template ClassToJavaGen or Implementation Choice 1, etc.

5. Related Work

Several approaches for managing variability in software product lines have been proposed in literature. A proposal for modeling variability in software families with UML using the standardized extension-mechanisms of UML is presented in [7]. A variation point model that allows user or application engineer to extend components at pre-specified variation points is proposed in [3]. A conceptual model for capturing variability in a software product line is presented in [1]. These three support the notion of variation point only and that too at modeling level. A general template-based approach for mapping feature models to concise representations of variability in different kinds of other models is presented in [5]. However it can address agility and adaptability of F and P dimensions only. The same concept is used in generation of model-based code generators from their high-level specifications [14].

Use of product line concept in model-based code generator is not new. It has been reported in several MDD literature [4, 7, 9, 15] to address the agility and adaptability of D, A and T dimensions. These approaches are based on aspect-oriented technique [2] and implemented using declarative language for model-to-text transformation [8]. The essential limitation of AOP based approach is that it works with the languages that support aspect weaving as this technique generates code fragments that need to be weaved for deriving deployable artifacts whereas, our solution is based on OO techniques, uses an imperative language and feature model notation, and does not require special composition machinery. It uses parameterization and composition techniques to derive a consistent code generator similar to traditional MDD code generators that generate deployable code.

6. Conclusion

We are working toward a framework for developing adaptive business application product line using model driven development techniques. Use of model driven techniques for developing such business applications is argued as a preferable option as platform independent specification can be retargeted to technology platform of choice through a code generation process. In our experience, use of product line architectures enables agility and adaptability in business functionality and business processes, i.e. F and P dimension of the business application for typical model-driven approaches. In this paper we argued the use of product line concept in F and P dimensions is not sufficient to achieve the
desired agility and adaptability - it is as well critical to addresses D, A and T dimensions. We presented an approach for developing configurable and extensible code generator product line. A meta model for specifying the product line and suitable extension to MTT specification for describing transformation rules in modularized form are presented. We demonstrated how a model (instance of code generator meta model) and transformational unit specification can be configured to derive a purpose specific code generator through an illustrative example.

We have used this approach to specify our MDD code generator product line resulting in time and effort savings [10]. Supporting a new deployment environment is now mostly a configuration activity rather than implementing a new code generator. Supporting new features in code generator family is much simplified process as one has to define transformational unit for the new feature with suitable associations with existing features and transformational units. Early observations of testing the approach on real life industrial strength problems are encouraging but there is still some way to go.

7. Reference


