Raising family is a good practice

Vinay Kulkarni
Tata Consultancy Services
54B, Industrial Estate, Hadapsar
Pune, 411013 INDIA
+91 20 66086301
vinay.vkulkarni@tcs.com

ABSTRACT
The need for adaptiveness of business applications is on the rise with continued increase in business dynamics. Code-centric techniques show unacceptable responsiveness in this dynamic context as business applications are subjected to changes along multiple dimensions that continue to evolve simultaneously. Recent literature suggests the use of product line architectures to increase adaptiveness by capturing commonality and variability to suitably configure the application. Use of model driven techniques for developing business applications is argued as a preferable option because platform independent specification can be retargeted to technology platform of choice through a code generation process. Business applications can be visualized to vary along five dimensions, namely, Functionality (F), Business process (P), Design decisions (D), Architecture (A) and Technology platform (T). Use of models is largely limited to F and P dimensions in commonly used model-driven development techniques thus limiting the benefits of product line concept to these two dimensions. We argue this is not sufficient to achieve the desired adaptiveness, and it is critical to extend the product line concept to D, A and T dimensions also. To address adaptation needs of business applications, this paper presents a model-driven generative approach that further builds on the ideas of separation of concerns, variability management and feature modeling. Early experience and lessons learnt are discussed, and future work outlined.

Categories and Subject Descriptors
D.2.m [**Software Engineering**]: Miscellaneous – *reusable software*.

General Terms
Management, Economics, Human Factors, Standardization, Languages

Keywords
Commonality, Variability, Adaptiveness, Model-driven development, Business applications, Product lines, Product families

1. INTRODUCTION
Rapid evolutions of technology platforms and business demands have contributed to significant increase in business dynamics in recent years. The increased dynamics put 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) software-intensive 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. This is an expensive and error prone process demanding large teams with broad-ranging expertise in business domain, architecture and technology platforms. Large size of an enterprise application further exacerbates this problem. Model-driven development alleviates this problem to an extent by automatically deriving an implementation from its high-level specification using set of code generators [20]. However, the scattering and tangling is the principal obstacle in agile adaptation of existing implementation for the desired change. Product line architectures aim to increase adaptiveness by capturing commonality and variability to enable application configurability. As the use of models is limited to F and P dimensions in commonly seen model-driven development techniques, the benefits of product line concept are also limited to these two dimensions. Therefore, it is critical to extend the product line concept to D, A and T dimensions also.

We present a model-driven approach that addresses adaptations needs along all the five dimensions using a specification-driven generative approach [8]. F and P dimension meta models are extended to support modeling of variability that is specified using feature model techniques. A meta model connecting these models

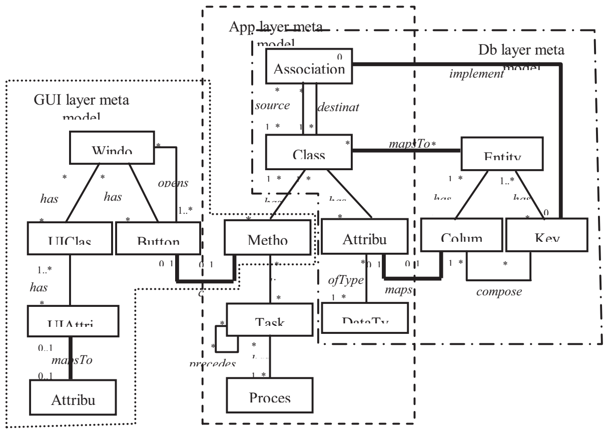

Fig. 1 Unified meta model for business application

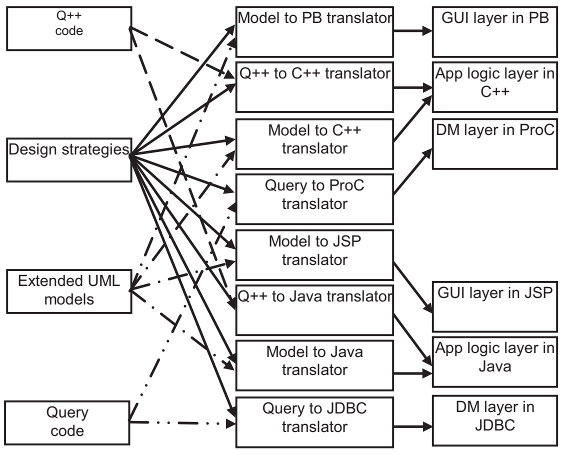

Fig. 2 Scattering and tangling in code generation

to the features in the feature model is presented. Building block abstraction as a means to implement a D or A or T feature is presented. A MDD code generator can be modeled as a hierarchical composition of building blocks, and maps easily to a consistent well-formed configuration of a feature model along D, A and T dimensions.

Section 2 describes model-based techniques we developed to automate development of business applications, and discusses our experience in using this approach to develop several large business applications on a variety of technology platforms and architectures. Section 3 presents an abstraction for organizing model-based code generators as a hierarchical composition of reusable building blocks, and discusses our experience and lessons learnt. Section 4 describes extensions to application specifications so as to model an application family. Section 5 discusses some of the related work. Section 6 concludes with a brief summary of early use of the proposed approach, and outlines future work.

2. GENERATING BUSINESS APPLICATIONS FROM MODELS

A typical database-intensive business application can be seen as a set of services with each service delivering specific business intent. These applications are best implemented using a layered architecture paradigm wherein each layer encapsulates a specific kind of data manipulations e.g. database access layer implements functionality such as primary-key based create/modify/get/delete and complex data accesses like joins, user interface layer implements how the data should be displayed on a screen and how the user will interact, etc. Thus, a set of code patterns recur in the implementation of an architectural layer. An architectural layer interfaces with its adjoining layer through a priori well-defined protocol. Thus, execution of a business application can be seen as an assembly-line of architectural layer specific processors that manipulate the work item, i.e. data corresponding to input and output parameters of a service, in a pre-defined manner before passing it over to the next processor in the assembly-line. As the processing is data-centric and a priori known, it can be easily generated for a given data definition.

Model-driven development approach starts with definition of an abstract specification that is to be transformed into a concrete implementation on a given target architecture [18]. The target architecture is usually layered with each layer representing one view of the system e.g. Graphical User Interface (GUI) layer, application logic layer and database layer. The modeling approach constructs the application specification using different abstract views- each defining a set of properties corresponding to the layer it models. We decompose an application specification into three such models- GUI layer model, Application layer model and Db layer model as shown in Fig. 1. We consider three meta models, namely GUI layer meta model, Application layer meta model and Db layer meta model, for the three view specifications. Each models views of a single Unified meta model as depicted in Fig. 1. Having a single meta model allows to specify integrity constraints to be satisfied by the instances of related model elements within and across different layers. This enables independent transformation of GUI layer model, Application layer model, and DB layer model into their corresponding implementations namely GUI layer code, Application layer code and Db layer code with assurance of integration of these code fragments into a consistent whole. These transformations are performed using code generators. The transformations are specified at meta model level and hence are applicable for all its model instances. If each individual transformation implements the corresponding specification and its relationships with other specifications correctly then the resulting implementations will glue together giving a consistent implementation of the specification.

2.1 Experience and lessons learnt

UML [16] modeling helped in early detection of errors in application development cycle. We associated with every model a set of rules and constraints that defined validity of its instances. These rules and constraints included rules for type checking and for consistency between specifications of different layers. We kept the models independent of implementation technology so as to be able to retarget them to multiple technology platforms i.e. gui platform, middleware, programming language, rdbms and operating system. We defined a higher level domain-specific language to specify business logic [10]. Non-primary key based

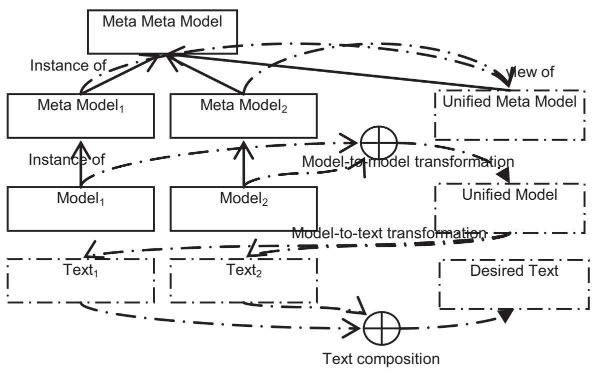

Fig. 3. Model-driven code generation architecture

complex data accesses were specified using a SQL-variant. Application specified in terms of models, Q++ and SQL-variant language was transformed into the target technology platforms encoding the chosen design strategies and architectural choices through a set of code generators. We preserved the divide and conquer strategy by having a code generator each for architectural layers.

Automated code generation resulted in significantly higher productivity in terms of lines of code [22]. Moreover, encoding of design strategies, guidelines and best practices into the code generators resulted in uniformly high code quality. Generation of interface code between the various architectural layers ensured smooth integration of independently generated code artifacts. We discovered that design strategies and architectural choices for no two applications were exactly alike necessitating development of application-specific code generators. Moreover, many architectural and design strategies cut across the layers. This required each tool to be aware of these cross cutting aspects. As a result, customizing for such cross cutting aspects required consistent modifications to several tools leading to maintenance problems. Increased acceptance of the approach led to the ironical situation wherein productivity toolset team became bottleneck for application delivery [21].

3. ORGANIZING MULTIPLE TOOLSETS INTO A FAMILY

As can be seen from figure 2, different code generators are needed to deliver the same business functionality on different technology platforms. This is despite these platform-specific code generators sharing a great deal of common functionality and mostly differing only in the use of primitives offered by the target technology platform e.g. syntax differences of programming languages, data type differences of databases, etc. Even while delivering identical business functionality on identical technology platforms, need to deliver onto different architectures e.g. synchronous, queue-based messaging etc demands different code generators. Similarly, use of different design strategies demands different code generators.

Thus, domain of model-based code generators can be described as a feature diagram [9] where intermediate nodes denote the variation points along D, A and T dimensions; the leaf nodes denote choices available for each variation point i.e. variations; and dependency between variation points expressed in terms of conditional expressions over their respective variations. The desired model-based code generator is a valid configuration over the feature diagram. However, feature diagram is just a declarative specification in terms of labels, and needs support for tracing a feature (i.e. the label) to its implementation. Recommended way for implementing the desired feature configuration is an ordered composition of the implementations of the set of constituent features. But, strict order is not always possible for inter-dependent features.

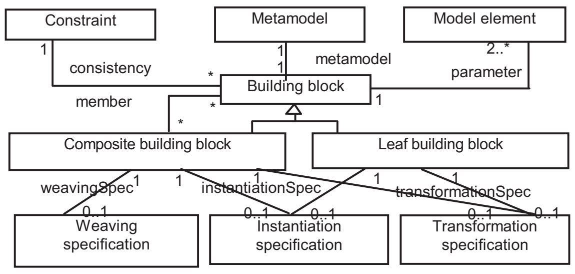

Fig. 4. Building block meta model

The tangling of model-based code generators, as shown in figure 2, is due to lack of separation of the various concerns, namely, technology platform, architecture and design strategies, and the cross-cutting nature of design strategies. An improved architecture for model-based code generation is where the models are successively refined through application of the various design strategies to a stage from where a platform specific implementation can be realized through a simple task of model-to-text transformation. As the platform specific code generators are independent of design strategy related issues, the same model-to-text transformation specifications can be reused with different design strategies and vice versa. This separation of concerns enables a tool variant to be viewed as a composition of design strategy and technology platform aspects of choice.

3.1 Building block

Building block is an abstraction that provides a traceable path to implementation for a feature as per the generic model-driven code generation architecture as shown in fig. 3. A building block is localized specification of a concern in terms of concern-specific meta model, model to model transformation, and model to text transformation. Building blocks are composable, enabling a model-driven code generator to be organized as a composition of a set of reusable building blocks, each encapsulating a specific concern. Figure 4 shows the building block meta model. A building block is essentially a means for expressing how a concern specification is transformed into models and code.

A model-driven code generator is specified as a hierarchical composition of building blocks of interest. Building blocks are of two kinds: *leaf building block* and *composite building block*. The instantiation specification of a leaf building block specifies how to stamp out the model elements of the unified model from the concern-specific model and the transformation specification captures how the model is transformed to code. We use QVT language [13] to specify the former and Mof2Text language [12] to specify the latter. A composite building block specifies how to compose its child building blocks. Weaving specification of a composite building block specifies how the code generated by its member building blocks is woven together. The process of model-driven code generation is realized through a post-order traversal

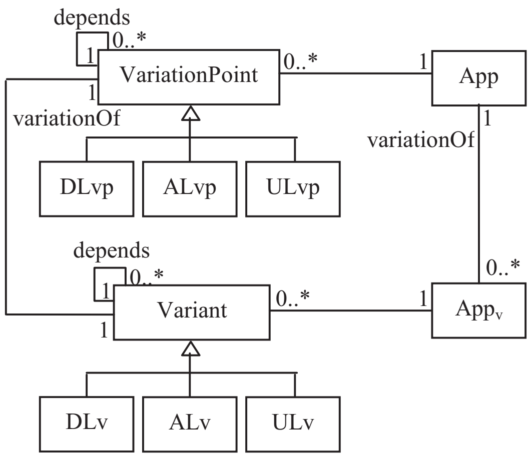

Fig. 5. A generic variability meta model

of the building block hierarchy in three sequential steps, namely, *Instantiation*, *Transformation* and *Weaving*. The instantiation step stamps out models and merges them. The transformation step transforms models into code snippets and generates weaving specifications for composing them. The weaving step composes the generated code snippets by processing the weaving specifications.

We translate a model (M_u) that is an instance of a unified meta model (MM_u) to various software artifacts like Java code, JDBC code, JSP code and a variety of configuration specifications in XML. Limiting aspect weaving only to code level artifacts would necessitate specialized weavers for Java, JDBC, JSP, XML etc. each having separate join point models. Also, this approach would necessitate some commonality over these join point models so as to have an integrated Java application. With increased number of software artifacts to be produced the approach becomes increasingly complex as essentially it amounts to building aspect infrastructure for each such artifact. We address this problem by specifying aspect weaving at the unified meta model level and performing it at the model level whenever possible. Unified meta model enables specification of relationships between the various (sub) modeling languages. A reflexive meta modeling framework provides the necessary infrastructure to define and integrate the various modeling languages of interest and a meta model aware model transformation framework provides the necessary technology to address model weaving requirements [10]. Performing aspect weaving at the model level also, whenever possible, results in reuse of model based code generators such as model-to-Java, model-to-JDBC, model-to-JSP and model-to-XML as these code generators are specified at the unified meta model level.

Multiple variants of a code generator realized as different compositions of building blocks can be easily organized into a family as follows,

- Commonality across variants can be specified using a set of common building blocks
- Functionality specific to each variant can be specified using a set of variant building blocks
- Composable nature of building blocks enables realization of the desired family member as a composition of suitable common and variant building blocks

Thus, building block abstraction provides a traceable path for a feature towards its implementation.

3.2 Experience and lessons learnt

Use of building block abstraction has enabled our toolset to be organized as a family or a product line wherein a tool variant can be easily composed from design strategy and technology platform aspects of choice. Containment of change impact due to localization and increased reuse due to composability have led to quick turnaround time for delivering a tool variant [17]. Use of a higher-level model-aware transformation language has made maintenance and evolution of the product line easy [14]. Also, building block abstraction has enabled us to organize the development team along two independent streams, namely, technology platform experts and design experts.

4. MODELING APPLICATION FAMILIES

Our organization discovered that solutions being delivered to different players in the same business domain were not exactly alike even for identical business intent. With toolset providing no means to capture commonality and variability, application development teams had to resort to copy-paste. As a result, what should have been a variant of an application ended up being a separate application thus leading to maintenance and evolution problems. These problems compounded with every new solution being delivered.

The approach described in section 2 generated a layered application implementation from a similarly layered specification. Each layer of application specification is an instance of its specific meta model. Now we describe how each of these meta models is enhanced to support the family concept [3]. Figure 5 depicts a meta model for capturing variability in a generic way. An application is viewed as a set of a priori defined *variation points* that could possibly be inter-dependent. Possible *variants* for each variation point are identified. Inter-dependence of variation points translates to similar relationship between their variants. Since our interest is database-intensive business applications that are typically implemented using a layered architecture, we identify variation points for each architectural layer. For instance, *DLvp* denotes variation points in database access layer, *ALvp* denotes variation points in application layer, and *ULvp* denotes variation points in user interface layer. Similarly, *DLv*, *ALv* and *ULv* denote variants in database access, application, and graphical user interface layers respectively. A set of *DLv* that honour dependency constraints between *DLvp*, a set of *ALv* that honour dependency constraints between *ALvp*, and a set of *ULv* that honour dependency constraints between *ULvp* constitute a complete, well-formed and consistent application variant App_v.

4.1 Approach

4.1.1 Application layer

Application layer specifies the business logic in terms of *Class*, *Attribute* and *Operations*. Being an encapsulation of both structural and behavioral aspects, Class is the natural choice for supporting the family concept in the application layer. Figure 6

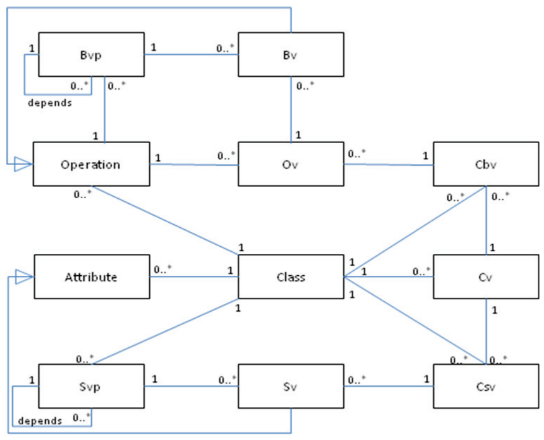

Fig. 6. Application layer meta model extension

depicts extensions to the application layer meta model highlighted in Fig 2 as follows:

- *Svp* denotes a structural variation point wherein multiple variations can be plugged.
- *Sv* denotes a structural variation for a structural variation point.
- A structural variation is in fact an *Attribute.*
- *Csv* denotes a structural variant for a *Class*. It is a complete and consistent configuration of structural variations of the class i.e. no structural variation point is left unplugged and selected structural variations honour structural variation point dependencies. Structural variants for a class differ in terms of the number of *Attributes* or their *Types* or both.
- *Bvp* denotes a behavioural variation point for an Operation wherein multiple variations can be plugged.
- *Bv* denotes a behavioural variation for a behavioural variation point.
- A behavioural variation is in fact an *Operation*.
- *Ov* denotes an Operation variant. It is a complete and consistent configuration of behavioural variants i.e. no behavioural variation point is left unplugged, and selected behavioural variants honour behaviour variation point dependencies if any.
- *Cbv* denotes a behavioural variant for a *Class*. It is a consistent configuration of Operation variants i.e., the selected operation variants serve meaningful intent.
- *Cv* denotes a variant for a *Class*. It is a type-compatible configuration of structural and behavioural variants of the class.

Thus, the above extensions enable modeling of a family of classes wherein each member (of the class family) serves the same intent in a specific situation. By making the above information available as metadata, implementation can switch from one consistent configuration of variants to another at application run-time. Not all such situational adaptations can be handled at application-runtime though, for instance, addition of a new behavior extension (*Bv or Ov*) would need recompilation (followed by redeployment). Similarly, definition of a new class altogether, as an extension to existing functionality, cannot be handled at application run-time. But, the meta model enables a new situation to be addressed such that it adds to the existing set of configuration alternatives.

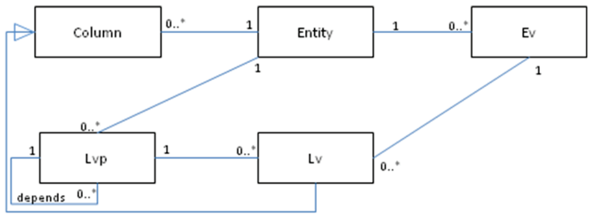

Fig. 7. Database layer meta model extension

4.1.2 Database layer

Database layer provides persistency to the desired application objects. We use object-relational database layer that provides an object façade to relational database tables implementing persistence. As an application object can vary *structurally*, the database table onto which it maps also needs to cater to this variance. And the same holds for structural extension as well. Configurability in database layer means quick switching from one known situation (i.e. db schema) to another, and extensibility means easy handling of as yet unseen situation. Figure 7 shows extension of database layer metamodel highlighted in fig. 2 as follows:

- *Lvp* denotes a structural variation point wherein multiple variations can be plugged.
- *Lv* denotes a structural variation for a structural variation point.
- A structural variation is in fact a *Column*.
- Ev denotes a structural variant for an *Entity*. It is a complete and consistent configuration of structural variations of the Entity i.e. no structural variation point is left unplugged and selected structural variations honour structural variation point dependencies. Structural variants for an Entity differ in terms of the number of *Columns* or their *Types* or both.

Thus, the above meta model enables modeling of a family of entities wherein each member (of the entity family) serves the same intent in a specific situation. In essence, the above information constitutes a generic db schema that can be specialized for a variety of situations. Database access methods such as primary-key based Create, Update, Get and Delete, complex data accesses like joins can encode interpretation of this information in their implementation. By making the above information available at application runtime, as metadata, implementation can switch from one known configuration to another at application run-time. Addition of a new row in the metadata tables corresponds to the ability of handling as yet unseen situation. Not all situational adaptations can be handled at application-runtime though, for instance, deletion of a column would need redefinition of the db schema leading to recompilation of database access layer code followed by redeployment. But, the meta model enables a new situation to be addressed such that it adds to the existing set of configuration alternatives.

4.1.3 User Interface layer

A GUI screen family represents a set of GUI screens that have a lot in common but differ from each other in a well-defined

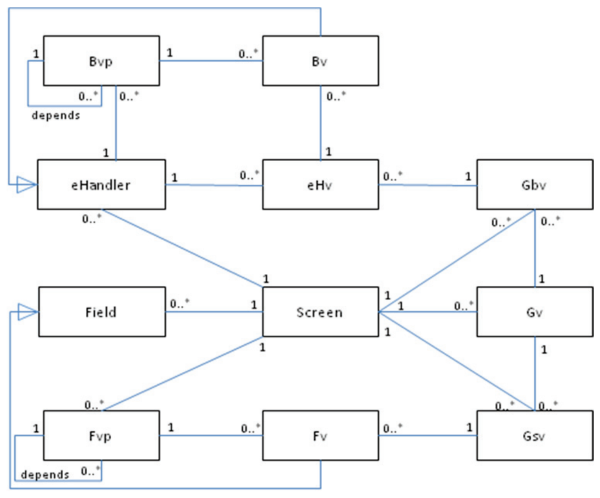

Fig. 8. User interface layer meta model extension

manner. Therefore, understanding of commonality and variability is critical in order to support modeling of a GUI screen family. A GUI screen is one of the channels for users to interact with an application. In essence, a GUI screen enables user to provide input data for carrying out a logical unit of work and display the response. While serving the same business intent, a GUI screen can vary in terms of *what* (i.e. data to enter and/or view) and *how* (i.e. layout information and GUI controls to use) leading to multiple situations. Configurability means quick switching from one known situation to another, and extensibility means easy handling of as yet unseen situation. Figure 8 shows an extension of user interface meta model highlighted in fig. 2 as follows:

- *Fvp* denotes a structural variation point wherein multiple variations can be plugged.
- *Fv* denotes a structural variation for a structural variation point.
- A structural variation is in fact a *Field.*
- *Gsv* denotes a structural variant for a *Screen*. It is a complete and consistent configuration of structural variations of the screen i.e. no structural variation point is left unplugged, and selected structural variations honour structural variation point dependencies. Structural variants for a screen differ in terms of the number of *Fields* or their lay-out or both.
- *Bvp* denotes a behavioural variation point for an event handler wherein multiple variations can be plugged.
- *Bv* denotes a behavioural variation for a behavioural variation point.
- A behavioural variation is in fact an *Event handler*.
- *eHv* denotes an event handler variant. It is a complete and consistent configuration of behavioural variants i.e. no behavioural variation point is left unplugged, and selected behavioural variants honour behaviour variation point dependencies if any.
- *Gbv* denotes a behavioural variant for a *Screen*. It is a consistent configuration of event handler variants i.e. the selected event handler variants serve meaningful intent.
- *Gv* denotes a variant for a *Screen*. It is a type-compatible configuration of structural and behavioural variants of the screen.

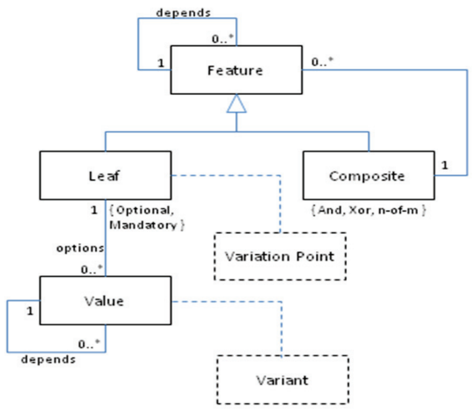

Fig. 9. Variability resolution meta model

Thus, the above meta model enables modeling of a family of screens wherein each member (of the screen family) serves the same intent in a specific situation. By making the above information available as metadata a GUI implementation can switch from one known situation to another at application run-time. Addition of a new row in the metadata tables corresponds to the ability of handling as yet unseen situation. Not all situational adaptations can be handled at application-runtime though, for instance, change in event handler code would need recompilation (followed by redeployment). Similarly, definition of a new screen altogether, as an extension to existing functionality, cannot be handled at application run-time. But, the meta model enables a new situation to be addressed such that it adds to the existing set of configuration alternatives.

4.1.4 Putting the layers together

Meta models described so far, in essence, help model a *family* at each architectural layer such that all members of a family share a *common* part and are distinguishable in terms of member-specific part. As described in section 2, the unified meta model enables specification of well-formedness constraints spanning across the architectural layers. Once a set of desired members, one from each architectural layer, is identified, it is possible to compose them into a well-formed specification that can be automatically transformed into a consistent and complete solution. We use feature modeling technique to enable selection of a family member pertaining to the desired criterion. Figure 9 shows the variability resolution meta model that captures traceability of a feature to its implementation artefacts. Augmented with constraints, this meta model enables selection of a configuration as a set of variants that is complete, consistent and well-formed. A configuration is:

- *complete* if there is no variation point remaining unplugged
- *well-formed* if structural constraints between variation point and variants being plugged therein such as Xor, And, n-of-m are satisfied
- *consistent* if variants honour dependency constraints between their variation points

In our experience this simplistic variability resolution meta model has sufficed so far. We are aware it will need to be richer going forward.

4.2 Experience and lessons learnt

We are in the early roll-out stage of this solution. In the interest of time-to-market, product owners decided to use the latest product release as baseline for introducing the family concept as opposed to refactoring the set of solutions delivered so far into a productline. Since new meta models are essentially an extension of old meta models, it was possible to migrate the older application models fully automatically. Given the simple nature of meta model extensions, full power of QVT [13] was not called for and a simpler imperative model transformation alternative [14] sufficed. In early experience, modeling of commonality and variability rooted at meta objects being used for code generation, namely, *Class*, *Entity*, *Operation* and *Screen* seem to suffice. Proposed meta models specify pre-defined variation points, possible variations, and constraints over variation points. Configuration is a process of selecting appropriate variations so that all variation points are consistently plugged-in for an application. We supported this configuration process at three different stages of application development: design time, installation time and run time. Design time configuration is supported through model transformation and model merge techniques. The installation time and run time configuration is supported by generating appropriate metadata for all possible variations. Having separated business process concern from application functionality, we had to support the family concept for business process models as well [19]. We think the true test of the proposed meta models as regards configurability and extensibility will come in supporting inherently dynamic domain of financial instruments, insurance products etc.

5. RELATED WORK

The idea of addressing a set of related situations in an integrated manner is not new. Parnas was the first to argue for the need to design software for ease of extension and contraction thus leading for software families [3]. Usual practice is to parameterize the solution so as to address known situations. Several approaches for supporting parameterization through variability management have been proposed. Extending UML for modeling variability using standardized extension mechanisms of UML is presented in [11]. A variation point model that allows extension of components at pre-specified variation points is proposed in [6]. A conceptual model for capturing variability in a software product line is presented in [4]. All the three only support the notion of variation point and that too only at modeling level whereas we provide support for structural and behavioural levels. Aspect-orientation [5] is a technique for addressing separation of concerns with greater modularization and locality. However, implicit communication link between aspects and classes complicates the readability and comprehension of an aspect-based realization of variability architecture [1]. We circumvent this problem by generating pure OO implementation with aspects suitably woven in. Feature modeling is a popular mechanism to specify product lines [9]. Although a feature model can represent commonality and variability in a concise taxonomic form, features in a feature model are merely symbols. Mapping features to other models, such as behavioral or data specifications, provides a path towards implementation. A general template-based approach for mapping feature models to concise representations of variability in different kinds of other models is presented in [7]. We build upon this idea to connect a feature to its implementation artefacts i.e. structural and behavioural specification through a meta model. Despite years of progress, contemporary tools often provide limited support for feature constraints and offer little or no support for debugging feature models. An integration of prior results to connect feature models, grammars, and propositional formulas so as to allow arbitrary propositional constraints to be defined among features and enable off-the-shelf satisfiability solvers to debug feature models is presented in [2]. We build upon these ideas to ensure consistency of the selected feature configuration. The meta model connecting features with their implementation artefacts guarantees consistency, correctness and completeness of the implementation.

6. SUMMARY

We presented a model-driven generative approach to address adaptation needs of business applications. The approach builds further on the ideas of separation of concerns, variability management, feature modeling and generative development. We visualize business applications to vary along five dimensions, namely, Functionality (F), Business process (P), Design decisions (D), Architecture (A) and Technology platform (T). We address adaptation needs along all the five dimensions using a specification-driven generative approach. We extend meta models to support specification of variability along F and P dimensions. We presented a meta model that connects these specifications to features in a feature model. This bridge meta model enables traceability of a consistent well-formed feature configuration to its specification artefacts thus realizing a family of application specifications corresponding to the feature model. We presented building block abstraction as a mean to implement a D or A or T feature. MDD code generator is a hierarchical composition of building blocks, and maps easily to a consistent well-formed configuration of a feature model along D, A and T dimensions.

We discussed our experience in using model-driven techniques to build large business applications on a variety of architectures and technology platforms. Separating business functionality from technological concerns, and model-based code generation resulted in significant productivity and quality gains. Encouraged by these benefits, many large development projects also readily adopted the model-driven approach despite initial investment in learning how to model. This enthusiastic, and somewhat unexpected, acceptance of the approach led to an ironical situation of the productivity toolset team becoming a bottleneck. We overcame this problem through use of product line techniques in order to model the code generators as a family, and deriving a purpose-specific implementation therefrom. Thus, we could achieve scale through addressing customizability at family level instead of individual member level.

We discovered the same issue with business functionality i.e. solutions delivered to different players in the same business domain were not exactly alike even while addressing the same business intent. We shared early stage experience of modeling commonality and variability along F and P dimensions which seems encouraging.

Though the idea of bringing together separation of concerns, variability management, and feature modeling seems promising, there are several open issues:

– The meta model providing traceability from F and P features to their implementation specifications is rather simplistic.

– There should be support, preferably tool-aided, for unit testing a feature - it should be possible to specify test cases for a feature independently and compose the test cases to arrive at the system level test cases for the desired feature configuration.
– There should be tool support for intelligent debugging at feature level. A bug detected at code level should be traceable back to the feature specification.
– Hierarchical organization of features enforces an ordered traversal. Complex interdependence of features may impede strict order.
– It is not clear which facets of a system deserve to be modeled as building blocks. There is a need to investigate how the engineering aspects can be modeled and what the right kind of abstractions for modeling them are to satisfy the various 'ities' like maintainability, reusability etc. For instance, how does one model a design for better maintainability?
– Building blocks may overlap each other thus introducing an order of weaving. How does one ensure that properties of all building blocks hold after their weaving?
– Supporting separation of concerns using building blocks raises several tooling issues. The modeling tool should be extensible to support new modeling languages so as to define new aspect models and relate them to existing models. The model transformation tool should have adequate support for pattern matching and composition. The tool should support incremental reconciliation of models and scale up to cater to the demands of enterprise class applications.

In comparison to the existing literature, the proposed approach centered around meta models capturing commonality and variability in all dimensions of a typical database intensive business application seems more pragmatic for industry use. We are working on development of a component abstraction and algebra to support configuration and extension operators for these dimensions. Also, going forward we hope to ride piggy-back the technology advance in OSGi [15].

7. REFERENCES

[1] Alexander Nyßen, Shmuel Tyszberowicz, Thomas Weiler. Are Aspects useful for Managing Variability in Software Product Lines? A Case Study. Early aspects workshop at SPLC 2005.

[2] Don Batory. Feature Models, Grammars, and Propositional Formulas. Software Productlines, Volume 3714 of LNCS, pages 7-20, Springer, 2005.

[3] D L Parnas. Designing Software for Ease of Extension and Contraction. Proceedings of the 3rd ICSE, pages 264 – 277,1978.

[4] Felix Bachmann, Michael Goedicke, Julio Leite, Robert Nord, Klaus Pohl, Balasubramaniam Ramesh and Alexander Vilbig. A Meta-model for Representing Variability in Product Family Development. Software Product Family Engineering, volume 3014 of LNCS, pages 66-80, Springer, 2004.

[5] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Longtier and John Irwin. Aspect oriented programming. ECOOP'97 LNCS 1241, pp 220-242. Springer-Verlag. June 1997.

[6] Hasan Gomaa, Diana L Webber. Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model. Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9 - Volume 9.Page: 90268.3

[7] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. Generative Programming and Component Engineering, Volume 3676 of LNCS, pages 422–437. Springer, 2005.

[8] K Czarnecki and U Eisenecker, Generative programming methods, tools and applications, Addison-Wesley, 2000.

[9] K Kang, S Kohen, J Hess, W Novak and A Peterson, Feature-orientation domain analysis feasibility study, Technical Report, CMU/SEI-90TR-21, November 1990.

[10] MasterCraft – Component-based Development Environment. Technical Documents. Tata Research Development and Design Centre. http://www.tata-mastercraft.com

[11] M Clauß, I Jena. Modeling variability with UML. GCSE 2001Young Researchers Workshop, 2001.

[12] MOF Models to Text Transformation Language http://www.omg.org/spec/MOFM2T/1.0/

[13] MOF Query / Views / Transformations http://www.omg.org/spec/QVT/1.0

[14] OMGen Reference manual, version 1.5, Technical Document, Tata Consultancy Services, May, 2008

[15] OSGi - The Dynamic Module System for Java, http://www.osgi.org/

[16] UML Infrastructure 2.0 Draft Adopted Specification, 2003, http://www.omg.org/spec/UML/2.0/

[17] Souvik Barat and Vinay Kulkarni: Developing configurable extensible code generators for model-driven approach. 22nd International Conference on Software Engineering and Knowledge Engineering, July, 2010.

[18] Vinay Kulkarni, R. Venkatesh and Sreedhar Reddy. Generating enterprise applications from models. OOIS'02, LNCS 2426, pp 270-279. 2002.

[19] Vinay Kulkarni and Souvik Barat: Business Process Families using Model-driven Techniques. 1st International workshop on Reuse in Business Process Management, Sep, 2010. http://each.uspnet.usp.br/rbpm2010/program.htm

[20] Vinay Kulkarni, Sreedhar Reddy, An abstraction for reusable MDD components: model-based generation of model-based code generators. GPCE 2008: 181-1843.

[21] Vinay Kulkarni, Sreedhar Reddy: Introducing MDA in a large IT consultancy organization. APSEC 2006: 419-426.

[22] Vinay Kulkarni, Sreedhar Reddy: Model-Driven Development of Enterprise Applications. UML Satellite Activities 2004: 118-128