Model Driven Software Development

A practitioner takes stock and looks into future

Vinay Kulkarni

Tata Consultancy Services, Pune, India
vinay.vkulkarni@tcs.com

Abstract. We discuss our experience in use of models and model-driven techniques for developing large business applications. Benefits accrued and limitations observed are highlighted. We describe possible means of overcoming some of the limitations and experience thereof. A case for shift in focus of model driven engineering (MDE) community in the context of large enterprises is argued. Though emerging from a specific context, we think, the takeaways from this experience may have a more general appeal for MDE practitioners, tool vendors and researchers.

Keywords: Modeling, meta modeling, separation of concerns, model transformation, software product lines, model driven engineering workbench, model driven enterprise

1 Introduction

We are in the business of delivering custom business applications for various verticals such as banking, financial services, insurance, retail etc. This paper describes our journey in use of models and model-driven software development techniques since their emergence till date. Chronological sequence of the narration, we think, might help bring out progression of our understanding of the problem and also evolution of home-grown MDE technology. We then discuss capabilities of minimal tooling infrastructure for easy adoption of MDE by industry practice leading to effective use. Then we take a sneak peek at possible future uses of models in the context of large enterprises. The paper concludes with a summary.

2 Model driven software development

Way back in ’94, our organization decided to come up with a focused offering in banking and financial services space. As system requirements for the same functional intent such as retail banking, payments, securities trading etc., vary from one financial institution to another, it was felt that developing a shrink-wrapped product wouldn’t do. Instead, it was felt that developing a set of functionality components having high internal cohesion and low external coupling would be more pragmatic [1]. A relevant subset of predefined components would be suitably modified to deliver the desired business application. The jump start courtesy of components would shorten time to market, it was felt. The offering was to have object-oriented nature in line with market expectations. It was also felt that the offering should be so designed and architected as to keep pace with technology advance.

2.1 Ground reality

Experience in delivering business-critical software systems had led to a small team of technical architects having expertise in distributed architecture and relational databases. Though C++ had emerged as dominant object oriented programming language, programmers found its complexity daunting. Also, there was no proven method available for developing industry-strength OO applications back in early ‘90s. As a result, the immediate principal challenge facing management was *how to quickly make the large team of fresh developers productive?*

2.2 Eliminating accidental complexity

Business applications typically conform to a layered architecture wherein each layer encapsulates a set of concerns and interfaces with adjoining architectural layers using a well-defined protocol. For instance, *Graphical user interface layer* deals with which widget to use for displaying which data and what the layout of the screen should be; *Business process layer* deals with ordering of profess steps/tasks and who should be performing which task and when; *Application services layer* deals with which functionality

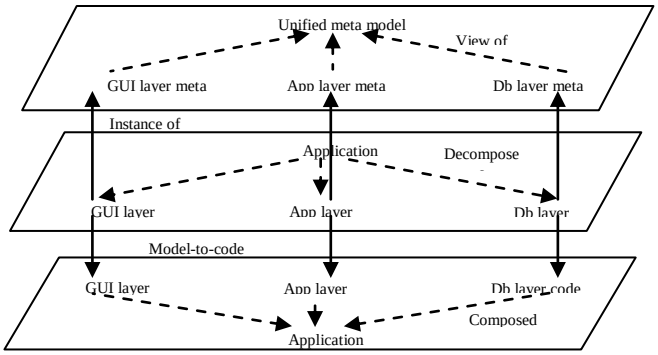

Fig. 1. Generating application from

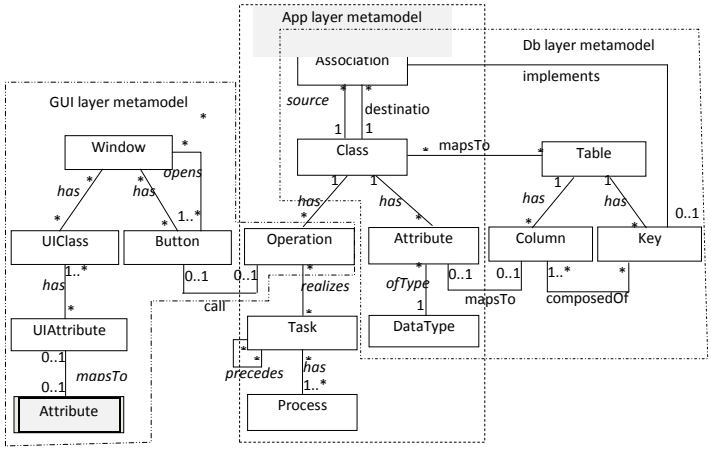

Fig. 2. A subset of the unified metamodel

to be exposed to the external world; and *Database access layer* deals with concerns such as how to construct an hierarchical object from flat tables and vice versa. Typically, the architectural layers are wired together by middleware infrastructure that support message passing in a variety of architectures such as synchronous, asynchronous, publish-subscribe etc. As a result, developing a distributed application demands wide-ranging expertise in distributed architectures and technology platforms which is typically in short supply. Large size of application further exacerbates the problem.

We addressed this problem through specification-driven code generation [2]. Keeping the layered nature of application in mind, we came up with domain specific languages (DSL), one for each layer, for specifying the application at a higher level of abstraction. Each DSL was a view over a Unified DSL which enabled specification of constraints spanning across two DSLs. For instance, a button on a screen must map to a service in the application services layer; a screen must have necessary data widgets so that input parameters of the service being invoked can be populated etc. By keeping the DSLs free from all technology related concerns, we helped developers focus on specifying just the business functionality. A DSL processor encoded appropriate technology related details while transforming a concern specification to the desired implementation [3]. We implemented these DSL transformers using standard compiler-compiler techniques [4]. Fig 1 describes the model-driven code generation approach pictorially.

We used UML [5] class diagrams to capture business entities and their relationships, UML use-case diagrams to describe business scenarios, and UML activity diagrams to describe process flows. We extended UML class models with additional properties and associations in order to capture architectural information, such as classes that make up a message, classes that need to be persisted in a database, classes that need to be displayed on a GUI screen, methods that need to be deployed as services having transactional behavior, class attributes that are mandatory etc. We designed a model-aware high-level language (Q++) to specify the business logic. Q++ treats the extended UML class models as its type system, provides constructs for navigating model associations, and allows for declarative specification of errors and exceptions. Also, the language abstracts out details such as memory management and exception handling strategy. Model-aware nature of Q++ guarantees that business logic specifications will always be consistent with the models.

We extended UML to specify the presentation layer in terms of special abstractions, namely windows and windowtypes. A windowtype specifies a pattern such as a form screen, a list screen, and so on. A window is an instance of a windowtype. It specifies which data elements of which business entity to display using which controls and which buttons should invoke which business services and/or open which windows. Our presentation layer model was independent of the implementation platform except for the event code.

We extended UML to specify relational database schemas and complex database accesses. Object-relational mapping was realized by associating object model elements to the schema model elements, i.e. class to table, attribute to column, association to foreign key etc. We defined a Query abstraction to provide an object façade over SQL queries with an interface to provide inputs to, and retrieve results from the query. We used a slightly modified SQL syntax in order to bind input/output parameters.

We came up with a unified meta model to specify the above mentioned models and their relationships. Fig. 2 highlights the associations spanning these models. These associations help keep the three specifications consistent with respect to each other and thus ensure that the generated platform-specific implementations are also consistent [2]. For example, the association Button.call.Operation can be used to check if a window is capable of supplying all the input parameters required by an operation being called from the window.

Similarly, we were also able to model other facets such as batch functionality, reports etc. From these various models and high level specifications, we were able to generate a complete application implementation.

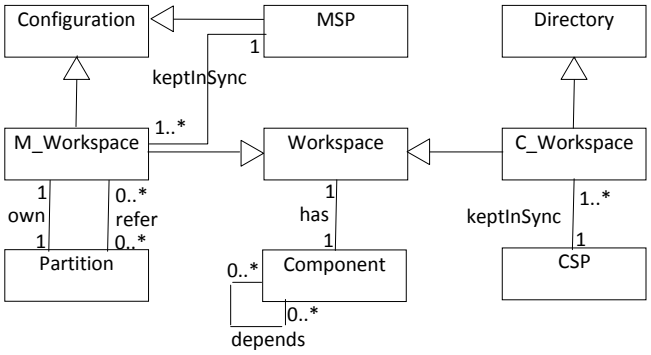

Fig. 3. Metamodel for workspaces

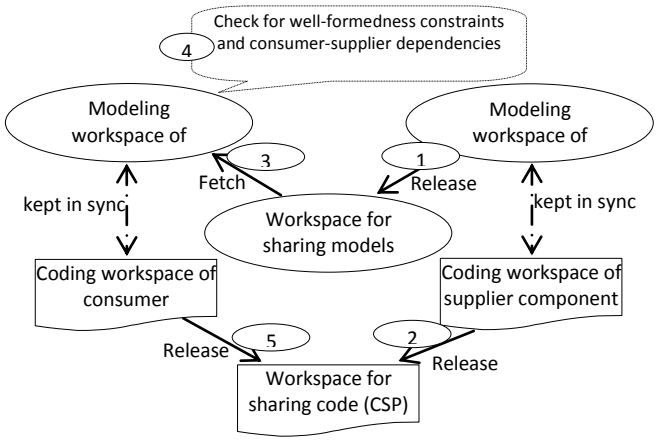

Fig. 4. Synchronizing components

Shifting the focus of software development from code to higher level of abstraction led to many advantages:

- Effective separation of concerns in specification provided a good handle on reducing the inherent complexity. DSLs being closer to the problem domain enabled functional experts to play a more significant and operational role in software development life cycle (SDLC). Higher level of abstraction also meant reduced size of the specification and hence reduced time for creating it.
- Greater structure courtesy of models led to application specifications being more amenable for rigorous analysis. As a result, certain kind of errors got caught early in SDLC and certain kind of errors were eliminated altogether. For instance, error of not mapping a screen button to an application service would otherwise get caught at the time of integrating independently developed screens with independently developed application services. In model-driven approach, this error gets caught at model validation stage itself before either the screen or the service is implemented.
- Model-driven development also helped in uniform application-wide implementation of policies such as date data type should always be displayed using, say, dateWidget control and for computation purposes be treated as a string conforming to "mmddyyyy:hh:mm:ss" format. In code-centric development, one has to rely on manual code inspection for enforcing such policies – an error prone and time- and effort-intensive endeavour. On the other hand, in model-driven approach such policies can be enforced either at model validation time or at model creation time itself.
- Code generators helped application-wide uniform implementation of key design decisions. For instance, query-intensive nature (as opposed to update-intensive) of application demands object-relational mapping strategy whereby the table corresponding to the derived class must have columns corresponding to the attributes of its base class all the way up to the root of the class hierarchy. This schema definition then dictates implementation of create(), get(), modify() and delete() methods which can be automatically generated from the class model. A change in design decision, say, switching over to update-intensive behaviour only needs transformation of the same database layer model using a different model transformer. On the other hand, correct and consistent implementation of this change would be a huge challenge for code-centric approach.
- We extended class and process models to capture testcase and testdata specifications for unit and system testing [6]. This helped us generate testdata with assurance of path coverage. Automation of test execution speeded up application testing process.
- Keeping application specification totally devoid of technology concerns, we could deliver the same specification into multiple technology platforms. This was possible largely because the target technology platforms had more or less similar capabilities. For instance, we could easily switch across databases (Oracle, DB2, Sqlserver), programming languages (C++, Java, C#), middleware (Tuxedo, CICS, Encina, Websphere), presentation managers (ASP, JSP, winforms) and operating system (Unix, Open MVS, Windows).

However, the shift to model driven development also raised some unique problems.

- Though application was specified at a higher level of abstraction, debugging remains at code level. As a result, one needs to carry in mind a map from specification to implementation for effective debugging. Higher the level of abstraction and the number of concerns abstracted, the more complex is the map and hence more difficult is the debugging.
- As modelling is not covered by majority of academic institutes as a part of their curricula, model-driven development had a steep learning curve for fresh developers who constituted a large portion of the project team.
- Modelling and model-based code generation tools needed to follow certain usage discipline. For instance, different concerns of the application being modelled independently needed to be validated for well-formedness and consistency –

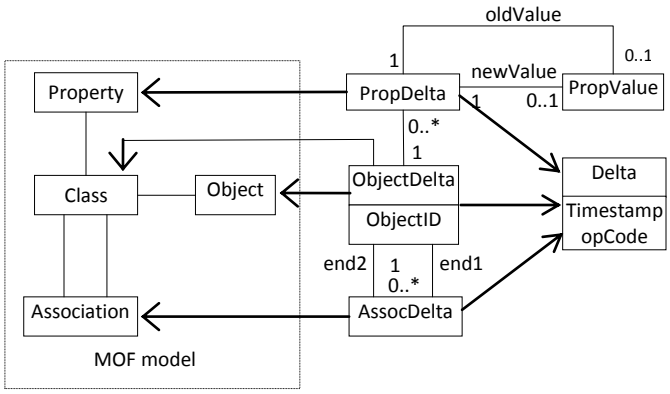

Fig. 5. Delta metamodel

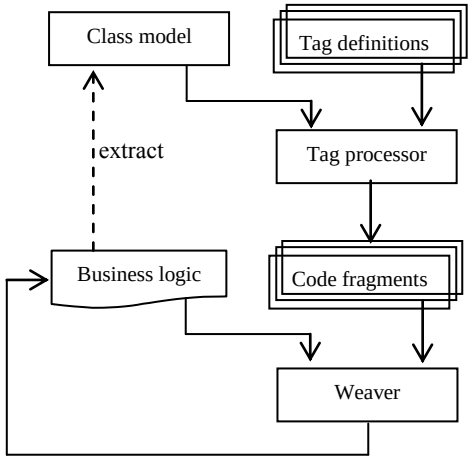

Fig. 7. Metadata-driven aspect-oriented development

both within a concern and across the concerns – as only valid models lead to generation of correct implementation in code. To cope with the large size, different concerns had to be modelled separately and synchronized on a need basis as concerns can be interrelated. This was a new way of software development for the project team. Just the availability of tools was not enough. Lack of a method for developing large applications using model-driven techniques proved to be the most significant hurdle in executing project in a coordinated manner.

- Onsite-offshore development model added a new dimension to the problem of coordinated project execution. Application specifications developed at different geographical locations needed to be synchronized from time to time.

With models and high level languages being the primary SDLC artefact, they needed to be amenable for versioning, configuration management, diff-n-merge etc.

2.3 Achieving scale-up

Component abstraction: Large business applications needed large development teams. Ensuring coordinated development with large teams became a big issue. Partitioning development effort along functional modules having high internal cohesion and low external coupling seemed intuitive. There was a need to make dependencies between these modules explicit so that independent development was possible.

We introduced a *component* abstraction as a unit of development to manage these dependencies at both model and code level. A component specifies its interface in terms of model elements such as *Classes, Operations and Queries*. The consumer-supplier relationship between components is explicitly modelled through *depends* association between the components. A component can only use the model elements specified in the interface of the components it depends upon. As Q++ is model-aware, these model-level dependencies can be honoured automatically in code as well. A component has two associated workspaces, a model workspace and a code workspace as shown in Fig. 3. The model workspace is a configuration comprising of own Partition and the Partitions of the components it depends on. In a component workspace, one is only allowed to change the contents of own Partition. As workspaces provide change isolation, a consumer component is not immediately affected by the changes introduced in its supplier components. A special workspace, configuration MSP (i.e. pool for sharing of models between components) of Fig. 3, is provided for exchanging models between components. A (supplier) component releases its model to this special workspace for sharing, from where its consumer components pick it up as shown in Fig. 4. Model well-formedness constraints and consumer-supplier dependencies are then automatically checked in the consumer component workspace. A similar workspace, directory CSP (i.e. pool for sharing of code between components) of Fig. 3, is provided for sharing code between components. Components are allowed to share code only after they share their models. Model-awareness of Q++ ensures consistency across consumer-supplier components at code-level as well. The process is realized through a set of roles, each responsible for performing a set of well-defined tasks on a component in a workspace. A role essentially identifies a set of logically coherent process steps. For instance, all modelling related tasks are grouped into the modeller role, all coding related tasks are grouped in the programmer role, all workspace synchronization related tasks are grouped in the manager role, all setup related tasks are grouped in the administrator role etc.

Explicit modelling of interfaces and dependencies provided better control over integration of components being developed independently and in parallel. This enhanced structure was used to compute change impact leading to significantly reduced testing effort. An extension of synchronization protocol of Fig 4 sufficed for multi-site development also.

Change-driven development: The performance of various model processing operations such as model validation, diff/merge, model transformation and code generation would deteriorate with increasing model sizes. This in turn would affect turn-around times for change management. Ideally, these operations should only consume time that is proportional to the size of the change and remain unaffected by the total size of the model. We devised a pattern-based approach for implementing incremental execution of common model-driven development process tasks such as model comparison and merging, model validation, and model transformation. We also developed a metamodel for recording changes and integrated it into the model repository for efficient change processing.

Models are well-structured graphs, and many model processing operations can be formulated in terms of graphs. Also, most models are processed in clusters of related elements; for instance, a class, its attributes and operations are usually processed together. Each such cluster has a primary element that identifies the cluster; for instance 'Class' in the above example. We used metamodel patterns to specify such clusters with the root identifying the primary element of the cluster. We then specified the model processing operations in terms of these patterns. For example, when we want to compare class models of two UML models, we want the comparison to be conducted on clusters of model elements centred around class objects; in pattern model terms we want to treat class as the primary object, with its attributes, operations and associations making up the rest of the connected elements of the cluster. In execution terms, a diff operation can be seen as being invoked repeatedly for each matching root element of the pattern from both the source and target models. Fig. 5 shows the metamodel for recording model changes that occur in a model repository.

A Delta is a record of a single change; it has a timestamp property that records the time of the change and an opCode property that records the operation causing the change, namely, one of ADD/MODIFY/DELETE. We developed a separate meta model, as shown in Fig. 5, so that changes to application models can also be captured in a model form. ObjectDelta records changes to objects; PropDelta records changes to properties; and AssocDelta records changes to associations. ObjectDelta has an association to Object to identify the object that has changed; it also stores the ID of the object (ID is required because that is the only way to identify an object that has been deleted from the repository). PropDelta has an association to Property to identify the property that has changed, and records two values – new and old (if any). AssociationDelta has an association to Association to identify the association that has changed, and two links to ObjectDelta corresponding to the two end objects. The associations between ObjectDelta, PropDelta and AssocDelta mirror the associations between Class, Property and Association in the meta meta model, and thus record the same structure. We devised an algorithm that, given a model pattern, computes the impacted root objects from a given model and its delta model for the set of changes recorded in a given time period.

Thus, we could identify which root objects in the model have changed in a given change cycle and apply the necessary model processing operations only on these root objects. This resulted in minimal code generation for the model changes. We used 'make' utility, which is time-stamp sensitive, to make the subsequent compilation → build → test → deploy steps also incremental.

2.4 Towards product-lines

In our experience, no two solutions even for the same business intent such as straight-through-processing of trade orders, back-office automation of a bank, automation of insurance policies administration etc., were identical. Though there existed a significant overlap across functional requirements for a given business intent, the variations were manifold too. Moreover, management expected delivery of subsequent solutions for the same business intent to be significantly faster, better and cheaper. We witnessed that business applications tend to vary along three dimensions:

- Functionality dimension which can be further divided into Business rules and Business logic sub-dimensions
- Business process dimension which can be further divided into Process tasks, Organizational policies, and Organizational structure sub-dimensions
- Solution architecture dimension which can be further divided into Design decisions, Technology platform, and Implementation architecture sub-dimensions

With code generators encoding the choices corresponding to the Solution architecture in transforming application specification into an implementation, we were forced to implement the code generators afresh for every project as no two projects had the same solution architecture even for the same business intent. We observed an interplay between the set of choices wherein a choice along a dimension eliminates (or forces) a set of choices along other dimensions. For instance, choice of 'rural India' geography for a banking system forced 'hosted services platform' choice of Implementation architecture; Choice of Java programming language and Oracle database as persistent store forced 'Object relational mapping' choice for design strategy etc. We witnessed that a choice along a dimension can impact multiple program locations (i.e. scattering) and choices along a set of dimensions can impact the same program location (i.e. tangling). For instance, choices for a set of strategies such as concurrency of a database table row (corresponding to a persistent object), object-relational mapping, and preserving audit trail in a database table all impact the create() method implementation for a persistent class. And object-relational mapping strategy impacts definitions of all persistent classes in the hierarchy.

These observations led us to visualize model-based code generation system as a set of composable Variable Units each having a set of well-defined Variation Points (VPs) as shown in Fig. 6. The variation points of a variable unit denote the places *where*

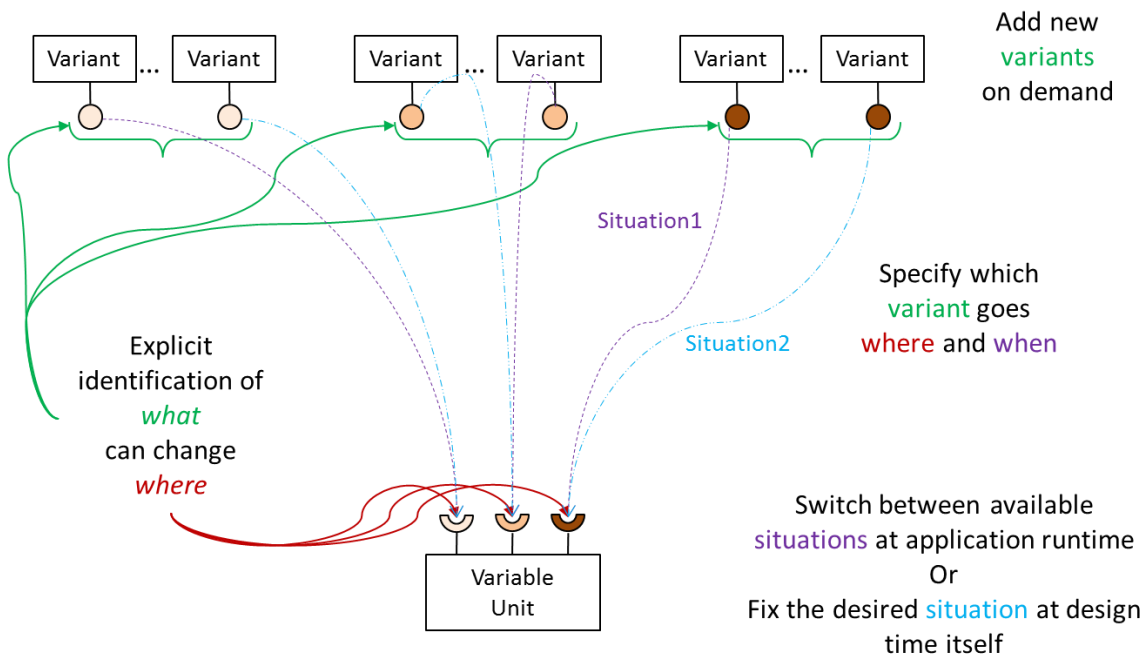

Fig. 6. Architecting for ease of configuration and extension

changes are expected to occur thus reflecting current level of understanding of the domain. A Variation (V) denotes *what* can change at a variation point so as to cater to a specific *Situation*. A *situation* helps to describe the context, i.e. *when* a specific change can occur. Addition of a new variation enriches *system configurability* i.e. ability to address more situations [7]. Also, the variation being added can have variation points of its own thus introducing new paths for extension and configuration. Thus, the system begins to take the shape of a product line wherein a member corresponds to a set of variable units and variations such that no variation point is left unbound, and the variations are *consistent* among themselves.

We came up with a modelling abstraction (*building block*) for specifying the architecture patterns of Fig 6 [8]. A building block encapsulates contribution of a given choice along Design strategy / Architecture / Technology platform sub-dimension to the eventual implementation. Thus, supporting new solution architecture is either novel composition of existing building blocks or addition of a new building block. This step towards model-based code generators product line led to several advantages:

- Model-based generation of model-based code generators reduced the time and effort required for maintaining the code generators.
- Building block abstraction facilitated reuse thus resulting in reduction in size of model-based code generator team.
- Separation of technology platform and MDE concerns led to specialization within code generator development team.
- Boot-strapping gave us confidence about functional completeness of our home-grown MDE infrastructure.

We are in the final stage of moving towards business application product line [10]. The core MDE infrastructure comprising of metamodels, model processors and a method for operationalizing a product line is in place. The central idea is vetted by implementing a model-based code generators product line. The product line idea has been proven in laboratory setting by implementing a near real-life example from Banking domain. We are about to start with a real-life product-line implementation through the restructuring and refactoring of a related set of existing purpose-specific solutions. We will be able to comment on robustness and usability of our MDE infrastructure only after completion of this exercise.

Early experience shows that models, through better separation of multi-dimensional concerns, seem to provide a better handle for implementing business application product lines. Separation of solution architecture from business functionality concerns enables business domain experts to focus solely on specifying the variations in business functionality and technology architects to focus solely on specifying the variations in technology platform, design strategies and implementation architecture. Ability to resolve the specified variations either at code generation time or at application run time leads to increased flexibility. Model-based generation of model-based generators leads to agile product line evolution process. The Variation Points also double up as extension points for introduction of as-yet-unforeseen changes – a reality for industry practice. Though early signs are encouraging, several significant issues remain to be addressed:

- Multi-level resolution of variability i.e. at class level, component level, application level etc., seems to suffice but is posing usability challenge even for the small laboratory case-study we implemented. In the least, more intuitive GUI seems a must for resolving variability.
- Business-critical applications need to evolve through extension and mutation. Our variability metamodel is adequate for addressing extension only i.e. add as-yet-unseen variant part or add as-yet-uncalled-for common part. But, a mutation may lead to fusion of existing variation points and commensurate fusion of a set of variant parts etc. Intuitive refactoring support is essential.
- Maintenance / evolution effort far exceeds the development effort for a successful business application [10]. Precise computation of impact of a change and optimal testing (i.e. what to test when) is a must.
- Effective management of a product line demands coordination of multiple stakeholders such as Domain experts, Solution architects, Developers, Testers, Product line managers etc., across the various SDLC phases. Should there be a feature model for every stakeholder? But a stakeholder might be interested in a set of [sub] dimensions leading to overlap of feature models and feature dependency. It calls for a method (and the relevant tooling) to help what to do when and by whom. There is a need to build further on the proposed multi-level resolution model and the staged resolution approach described in [11].

Definition of a new mutual fund offering or an insurance policy or a financial product varies from the existing ones in a well-defined manner even though the variations need to be introduced at many places. A declarative mechanism aided by suitable [de]composition architecture is missing.

2.5 Code is the model

Though the model-driven software development approach has delivered on the promises of improved productivity, better quality and platform independence, especially in development of large enterprise applications, majority of the small to medium sized projects found it difficult to adopt for several reasons. Steep learning curve for modelling meant a considerable chunk of project execution time was spent in creating application specifications. Insistence on models being the primary SDLC artefacts further exacerbated this problem. Project teams wanted bi-directional synchronization between model and the generated code for quick turnaround of changes introduced. An application generated from models is best maintained using the generators. This didn't augur well with many customers who were sensitive to technology and vendor lock-in related risks. Also any change in the solution architecture of the application to be delivered necessitated modifying the code generators which demanded expertise in MDE technology – a skill in excessive short supply. As a result, small and medium sized development projects found model-driven software development approach too heavy and the associated development process too restrictive.

To overcome these limitations, we came up with a metadata-driven aspect-oriented approach for developing J2EE applications (and later extended it for .Net applications). The key idea is to implement just the business logic (i.e. Functionality dimension mentioned earlier) using a reflexive and extensible programming language like Java / C# and annotate it with *tags* [12]. A tag encapsulates choice along one of the many dimensions of Solution architecture such that the desired solution architecture can be viewed as a hierarchical composition of tags. Annotated business logic is parsed to create an internal representation in the form of class model and its metadata annotations. The class model is transformed to generate skeleton or partial class definition. The tag hierarchy is transformed to generate a set of code fragments, each corresponding to the choice along one of the many solution architecture dimensions, and a specification for composing these code fragments. We used AspectJ [13] like syntax for specifying code composition and implemented a tree-transformation based composer which turned out to be sufficient for our needs. Fig 7 gives a pictorial description of this approach.

The approach was supported through an open extensible Eclipse-based toolset resulting in a development process that turned out to be flexible and lightweight as compared to model-driven development process. Separation of architect role (to specify tags) from developer role (to specify business logic and use pre-defined tags) led to effective utilization of expertise. The composable nature of building blocks enabled reuse even across projects. Template based code generation strategy enabled easy adherence to customer-specific standards, guidelines, best practices etc. The composable nature of building blocks, the ability to define purpose-specific metamodels and plug-in architecture due to Eclipse resulted in easy customizability, extensibility and improved maintainability of the toolset. The approach facilitated creation of a repository of reusable artefacts. A library of reusable, easy-to-adapt solution accelerators enabled even inexperienced teams to deliver high quality code on schedule. Low or no learning curve, adherence to industry standards and dependence on freeware further accelerated acceptance of the approach within developer community.

Class model centricity turned out to be a severe limitation of this approach. Other concerns such as business process, graphical user interface etc could not be addressed easily. In effect, it amounted to coming up with new join point models and developing suitable implementation machinery for every join point model which is a significant effort. Model-driven approach turned out better when multiple DSLs were needed for implementing the required solution. Therefore, we developed a mechanism that enabled interoperation between model-driven and metadata-driven approaches. The most observed use of this interoperability bridge was to graduate from metadata-driven approach to model-driven approach. Although possible, we didn't observe reverting back to metadata-driven approach after having settled into model-driven approach.

3 Tools for model driven development

Model-driven software development has been around since mid-90s. Launch of OMG's MDA [14] in 2000 generated widespread interest in model-driven development. Today it can justifiably be said that model-driven development has proved beneficial in certain niche domains if not all. There is ample evidence of models being used in many ways viz., as pictures, as documentation aids, as jump-start SDLC artefacts, as primary SDLC artefacts etc [15, 16]. Many tools exist that provide automation support at various levels of sophistication for model-driven development. The majority of these tools are highly effective when used in a shrink-wrapped manner but tend to lose effectiveness rapidly whenever the tool needs to be customized or extended. This is a serious drawback.

The demand for intuitiveness on models dictate they be domain-specific. Since there can be infinitely many domains with each domain possibly ever-expanding, it is impossible to think of a universal modelling language that can effectively cater to them all. Furthermore, models are purposive and hence it is impossible to conceive a single modelling language that can cater to all possible purposes. Therefore, multiplicity of modelling languages is a reality. Separation of concerns principle makes the need for a cluster of related modelling languages (one for each concern in a domain) and a mechanism to relate the separately modelled concerns (say to compose a unified model) apparent. The need to relate otherwise separate models demands an ability to express one model in terms of the other. Thus emerges the need for a common language capable of defining all possible modelling languages of interest. There are multiple stakeholders for a model each possibly having a limited view being presented in the form a suitable diagramming notation. From the above discussion, it follows there could be as many diagramming notations as there are modelling languages. And thus emerges the need for a language to define all possible visualizations of a model. For models to be used as primary SDLC artefacts, there needs to be an execution engine for the models – say an interpreter or a transformer to (say) text format that is executable e.g. a programming language. Plus, separation of concerns leading to a cluster of relatable models indicates the need for transforming one model into another and another and so on. Therefore, to comprehensively address the needs of model-driven software development the MDD tool needs to support:

- A language to define all possible modelling languages
- A language to define all possible visualizations of a model
- A language to specify transformation of one model into another
- A language to specify transformation of a model into text artefacts

As a result, software development gets transformed into a language engineering endeavour wherein the focus is on defining the most intuitive and expressive modelling language[s] for a given purpose and the necessary execution machinery. Since there cannot be a bound on the desired purposes, a configurable extensible modelling language engineering workbench seems required for greater adoption of model-driven software development. In a sense, something on the lines of Eclipse but for language engineering is called for [17].

4 What next

Economic and geo-political uncertainties are putting increasingly greater stress on frugality and agility for enterprises. Large size and increasing connectedness of enterprises is fast leading them to a system of systems which is characterized by high dynamics and absence of a know-all-oracle. Multiple change drivers are resulting in increasingly dynamic operational environment for enterprise IT systems, for instance, along Business dimensions the change drivers are dynamic supply chains, mergers and acquisitions, globalization pressures etc., along Regulatory compliance dimension the change drivers are Sarbanes oxley, HiPAA, Carbon footprint etc., and along Technology dimension the change drivers are Cloud, smartphones, Internet of things etc. At the same time windows of opportunity for introducing a new service/product/offering and/or for adapting to a change are continuously shrinking. Furthermore, business-critical nature of IT systems means the cost of incorrect decision is becoming prohibitively high and there is very little room for later course-correction. Therefore it is important that we look beyond the traditional model-based generative/SPLE based techniques that we have been using in the past and put more emphasis on understanding of the target organizational environment including its business, IT systems, and stakeholder perspectives. In other words, model the whole enterprise. Towards formal and precise enterprise architecture modelling is an important step towards realizing this goal.

Enterprise Architecture (EA) is a technique used in the process of translating business vision and strategy into effective enterprise change by creating, communicating and improving the key requirements, principles and models centred on business and IT that describe the enterprise's future state and enable its evolution [18]. A number of EA techniques are used by companies looking for business-IT alignment and transformation with desired properties, for instance, Zachman Framework, TOGAF (Open Group Architecture Framework), FEA (Federal Enterprise Architecture), Gartner, and ArchiMate, etc [19].

Applying these EA techniques to an enterprise is a highly person dependent activity with complete reliance on the enterprise architect's knowledge and experience. Furthermore, validation of goals, such as business-IT alignment, is carried out in a blue-print way in current EA techniques [20]. It means that if the enterprise architect feels, based on his knowledge and experience, that an

enterprise has been architected according to principles laid out by these EA techniques; then goals such as business-IT alignment have been accomplished *by definition*. An enterprise may also strive for other goals such as adaptability or cost optimality, for which no mechanism is provided by current EA techniques to prove that a property is satisfied across the enterprise. We believe the ability to specify enterprises in terms of high-level machine-manipulable models that are amenable to analysis and simulation is critical.

The more closely enterprise models reflect reality, the more applicable the inferences from simulation and analysis will be. Today, data describing structural and behavioural aspects of an enterprise is available – typically from multiple and possibly overlapping perspectives. We believe it should be possible to make use of this data to automatically arrive at first-cut purpose-specific models. Results from the well-studied field of log analysis seem readily applicable here [21]. These models will typically be further refined (and glued together into a unified model) by the experts. Results of what-if / if-what analysis and simulation of a purpose-specific model can help arrive at appropriate response to the problem under consideration. A model-centric approach will enable problems to be addressed in a pro-active manner long before they manifest. A mapping or a faithful representation from the models-for-analysis ("simulated" enterprise) to enterprise system models ("real" enterprise) is required to translate inferences from analysis and simulation into an action plan for the underlying enterprise systems. The action plan will typically describe a partially ordered list of changes to be introduced in the operating environment, or in software intensive systems, or in the manner the software intensive systems interact with each other and environment, or any combination of these.

5 Summary

We discussed our experience of delivering large business applications using model driven development approach supported by home-grown standards compliant MDD toolset. In our experience, large application development projects benefitted from this approach in terms of technology independence, enhanced productivity and uniformly high code quality. We observed that without a method imparting discipline to their use, the modelling tools and model-based code generators create more problems for the development team. As a consequence, model-driven development approach leads to less agile if not inflexible development process. We have tried incorporating agile manifesto in model-driven development but it is too early to say anything concrete [22].

Steep learning curve and high upfront investment in the form of creating high level specifications were principal deterrents for small and medium sized software development projects adopting model-driven approach. A metadata-driven code-centric generative approach turned out more appropriate. But, this approach turned out to be too limited to be considered for complex software development endeavours. To overcome this limitation, we came up with a lightweight model interpretation based approach [23]. This worked very well as long as non-functional requirements such as throughput, transaction time were not stringent. As these concerns were effectively addressed in model-driven code generative approach, we developed a migration bridge from interpretive to code generative approach. But this bridge was sporadically used and with mixed response which we haven't fully analysed yet.

Managers agreed to the qualitative benefits of model-driven development but inability to translate them in quantitative terms tended to be the biggest stumbling block for adoption of model-driven approach. We have taken a baby step in this regard but lot needs to be done [24].

We think the basic technological pieces for supporting model-driven development are in place. Many tools with a varying degree of sophistication exist. Other important aspects such as usability, learnability, performance need to be improved which in essence is a continuous process. However, full potential of model-driven development cannot be realized in absence of ready-to-use models supported by domain ontologies providing the semantic and reasoning basis. This aspect is poorly addressed at present.

Focus of MDE community has been on developing technologies that address *how to model*. Barring the domain of safety-critical systems, these models are used only for generating a system implementation. Rather, modelling language design/definition is influenced very heavily by its ability to be transformed into an implementation that can be executed on some platform. Modern enterprises face wicked problems most of which are addressed in ad hoc manner. Use of modelling can provide a more scientific and tractable alternative. For which, modelling community needs to shift the focus on analysis and simulation of models. Results from random graphs, probabilistic graphical models, belief propagation and statistics seem applicable here. We believe, it is possible to model at least a small subset of modern complex enterprises so as to demonstrate that model *is* the organization [25].

6 Acknowledgment

Author would like to acknowledge Sagar Sunkle, Suman Roychoudhury, Sreedhar Reddy and Tony Clark for direct and indirect help leading to this manuscript.

7 References

1. Paul C. Clements, "From subroutines to subsystems: Component-based software development." American Programmer 8 (1995), pp. 31-31.
2. Vinay Kulkarni, R. Venkatesh and Sreedhar Reddy. “Generating Enterprise Applications from Models”. OOIS Workshops 2002, pp. 270-279.
3. Vinay Kulkarni and Sreedhar Reddy. “Integrating Aspects with Model Driven Software Development”. Software Engineering Research and Practice 2003, pp. 186-197.
4. Aho, Alfred V., et al. Compilers: principles, techniques, and tools. Vol. 1009. Pearson/Addison Wesley, 2007.
5. UML – Unified Modeling Language, http://www.omg.org/spec/UML
6. Ashok Sreenivas. "Panel discussion: is ISSTA testing research relevant to industrial users?." ACM SIGSOFT Software Engineering Notes. Vol. 27. No. 4. ACM, 2002.
7. David L Parnas. “Designing software for ease of extension and contraction”. ICSE 1978, pp. 264-277.
8. Vinay Kulkarni and Sreedhar Reddy. "An abstraction for reusable MDD components: model-based generation of model-based code generators". GPCE 2008, pp. 181-184.
9. Vinay Kulkarni, Souvik Barat and Suman Roychoudhury. "Towards Business Application Product Lines". MoDELS 2012, pp. 285-301.
10. Barry Boehm. "A Spiral Model of Software Development and Enhancement", ACM SIGSOFT Software Engineering Notes 1986, 11(4), pp. 14-24.
11. K Czarnecki, S Helsen and U Eisenecker. “Staged configuration using feature models”. SPLC 2004, pp. 162-164.
12. Vinay Kulkarni. “Metadata-driven aspect-oriented software development”. Technical Architects Conference, 2004.
13. Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. "An overview of AspectJ." ECOOP 2001—Object-Oriented Programming (2001), pp. 327-354.
14. MDA – Model Driven Architecture http://www.omg.org/mda
15. Hailpern, Brent, and Peri Tarr. "Model-driven development: The good, the bad, and the ugly." IBM systems journal 45, no. 3 (2006), pp. 451-461.
16. Hutchinson, John, Mark Rouncefield, and Jon Whittle. "Model-driven engineering practices in industry." In Software Engineering (ICSE), 2011 33rd International Conference on, pp. 633-642. IEEE, 2011.
17. Vinay Kulkarni, Souvik Barat and Sagar Sunkle. “Model driven development – where to from here? A practitioner’s perspective. Advances in Model Based Engineering Workshop of India Software Engineering Conference 2012. http://www.infosys.com/infosys-labs/publications/infosyslabs-briefings/Pages/software-engineering-approaches.aspx
18. IEEE recommended practice for architectural description of software intensive systems http://standards.ieee.org/findstds/standard/1471-2000.html
19. R. Sessions. “A comparison of the top four enterprise-architecture methodologies”. 2007. MSDN- Enterprise Architecture Trends. http://msdn.microsoft.com/en-us/library/bb466232.aspx
20. R. Wagter, E. Proper and D. Witte. "A practice-based framework for enterprise coherence". PRET, 2012, pp: 77-95.
21. A. Margara and G. Cugola. "Processing flows of information: from data stream to complex event processing". DEBS 2011, pp: 359-360.
22. Vinay Kulkarni, Souvik Barat and Uday Ramteerthkar. “Early Experience with Agile Methodology in a Model-Driven Approach”. MoDELS 2011, pp. 578-590.
23. Gautam Shroff, Puneet Agarwal, Premkumar T. Devanbu: InstantApps: A WYSIWYG model driven interpreter for web applications. ICSE Companion 2009, pp. 417-418.
24. Sagar Sunkle and Vinay Kulkarni. “Cost Estimation for Model-Driven Engineering”. MoDELS 2012, pp. 659-675.
25. Tony Clark, Vinay Kulkarni, Robert France, Ulrich Frank and Balbir Barn. “Towards model driven organization”. NIER manuscript submitted to ICSE’13.

adfa, p. 11, 2011.