Model Driven Software Engineering

The model driven approach for software development

Mousami
B.E. Final Year- Information Technology
Department of Computer Science and Engineering
M.B.M. Engineering College, Jai Narain Vyas University
Jodhpur, India
Email: mousami.it@jnvu.edu.in

***Abstract*—Model Driven Software Engineering (MDSE) is the practice of designing a complex software system in terms of models and generating its code from them automatically. It provides abstraction of software development from the domain specific constraints of programming languages and conceives the software at the same level as its problem specification. This paper presents a brief study explaining the details and applications of model driven software development approach. It traces the history of Model Driven Engineering (MDE) in software engineering field and studies how the abstract models are created and converted into actual implementations. This paper also sheds light on how to create models using different level of abstractions, the basic techniques used for automated code generation using the models and the comparison of those techniques. A brief overview of the advantages and disadvantages of using MDE for software development given at the end of this paper concludes that MDSE is a major step towards the automation of coding and thus is leading to the industrialization of software development.**

Keywords—MDE; CASE; automated code generation; code generators

I. INTRODUCTION

The software systems of today are a lot more complicated than they were a few decades ago. The software in today's systems is often required to operate in distributed and embedded computing environments consisting of diverse devices and platforms and behave in a dependable manner. Due to the increased platform complexity, developing intricate and dependable software systems using code-centric technologies is a very tough job. One of the major factors for this difficulty in developing complex software is the huge gap between the problem and the domain of implementation. And trying to bridge this gap by techniques which require extensive handcrafting of implementations gives rise to increased accidental complexities which makes the development of complex software systems more difficult and costly. The growing complexity and the incompetency of the modern programming languages to bridge the implementation gap have generated the need of paradigm shift in software development thus leading to Model Driven Engineering (MDE) approach in software development. The paradigm shift brought about by Model Driven Software Development has become an increasingly popular approach to deal with the complexity of software engineering. In this model of software development, the aim is to develop models instead of source code. The level of abstraction is raised from source code to models.

The core idea of model driven approach is that it allows the coding to be done automatically from a set of well defined models as input.

II. HISTORICAL CONTEXT

A. Evolution of Abstraction

The notion of abstraction is not new in software engineering field. First the assembly languages were used for machine coding, thus making the complexities of machine code abstract. After that more advanced programming languages such as C raised the level of abstraction over assembly language. And now the next level of abstraction is modeling. A lot of work has been done in the field of models. A reference of models and their use in code generation was seen in the Computer-Aided Software Engineering (CASE) tools in the 1980s which provided other abstractions with some tooling.

B. CASE

Computer aided software engineering (CASE) was the most prominent effort that begun in the 80s[1] for raising the abstraction level of software programming. It emphasized on developing software methods and tools that helped developers to represent their designs in terms of general purpose graphical programming representations, which included state machines, structure diagrams, and dataflow diagrams. The major goals of CASE were as under:

- To enable more thorough analysis of the less complex graphical programs than the conventional general-purpose programming languages like C.
- To generate artifacts from graphical representations that can be used for implementation.
- To reduce the effort of manual coding, debugging, and porting of the programs.

Despite of the considerable attention in the research and trade literature, CASE was not widely adopted in practice. It's major failing points are described below:

- The general-purpose graphical language representations used for writing programs in CASE tools mapped poorly onto the underlying platforms.
- CASE could not handle complex systems having wide range of application domains.
- CASE tools did not support concurrent engineering, so they were limited to programs written by a single person or by a team that serialized their access to files used by these tools.
- There was a lack of powerful common middleware platforms for the CASE tools.
- CASE tools had "one-size-fits-all" graphical representations which were very generalized and could not be customized according to different application domains.

As a result of above failures, CASE had relatively little impact on commercial software development during the 1980s and 1990s, focusing primarily on a few domains, such as telecom call processing, that mapped nicely onto state machine representations[1].

The Model Driven Engineering approach which is used now-a-days has evolved past most of the above faults of its predecessor CASE and is hence the next big thing in software engineering field.

III. Model Driven Engineering in Software

MDE can be defined as- "the use of relevant **abstractions** that help people focus on key details of a complex problem or solution combined with **automation** to support the analysis of both the problem and solution, along with the mechanism for combining the information collected from the various abstractions to construct a system correctly"[2]. In other words, Model Driven Software Development consists of two parts:

- Defining models correctly for a complex problem and its solution with appropriate abstractions so that they focus on the important details relevant to the software.
- Providing automated coding from the models designed above through mechanisms that combine all the abstractions consistently and in accordance with the software to be generated.

IV. Models and Abstraction Levels

Models are the key artifacts in the MDE approach of software development. They are the basic ingredients for development of complex systems using MDE. They work in the domain of problem itself. For creating a model for code generation, we should first decide on the various levels of abstractions. Some of the key abstractions can be categorized into types which are described below:

- **Structure** (interfaces): abstraction is done on structural level. For example systems, subsystems, components, modules, classes, and interfaces (inputs and outputs). Only the structural details are shown in the model and rest details are abstracted.
- **Behavior** (functionality): in this category, only the functionality of the software to be generated is shown. Hence, all the behavioral details are presented in the model while other details are hidden or not shown.
- **Timing** (concurrency, interaction): in the models consistent with timing, all the details regarding concurrency protocols, interaction of processes with shared resources and the constraints regarding the real time execution of the software system are addressed.
- **Resources** (environment): the models represent the resources available to the software and its working environment.
- **Metamodels** (models about models): a very interesting category of models is the metamodel. They are models about the models, i.e., they specify, how a model should be designed for efficient code generation with platform independency.

Some of these abstraction concepts have existed and evolved with programming languages, but within a programming language the combination of these views is lumped or tangled together (e.g., spaghetti code)[2]. Although through good coding practices programs can be better structured and layered, but models help in systematically separating the different views. And since certain types of models are allowed to permit only certain types of information, even complex software systems can be broken into simpler models thus making the software more readable and maintainable, a feat that cannot be achieved through programming languages.

V. Generation of Code

MDE automation analyzes the views using automated means, derives the required information from one or more views, and then ultimately pulls sets of views together correctly to produce the specified complex software system using automatic code generator.

There are different ways how to design and implement a code generator. Following are several well-known generation patterns:

- Templates and filtering
- Templates and metamodel
- API-based generators
- Inline code generation

Different generation patterns have different advantages and disadvantages

A. Templates and Filtering

Templates and Filtering describe the simplest way of generating code. Code is generated by applying templates to textual model specifications (often XML/XMI), typically after filtering some parts of the specification. The code to be generated is embedded in the templates. Fig.1 presents a schematic of this technique of code generation.

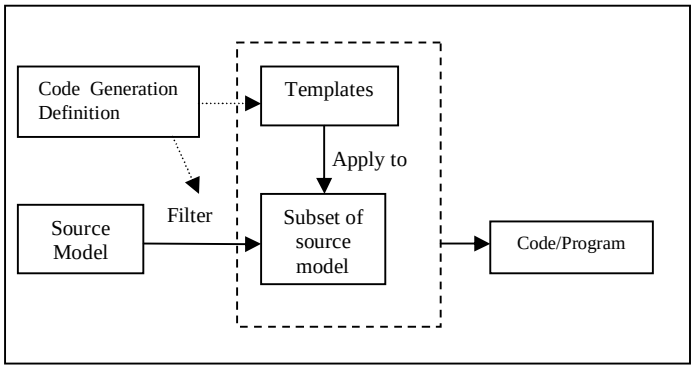

Fig. 1. Templates and Filtering code generation technique

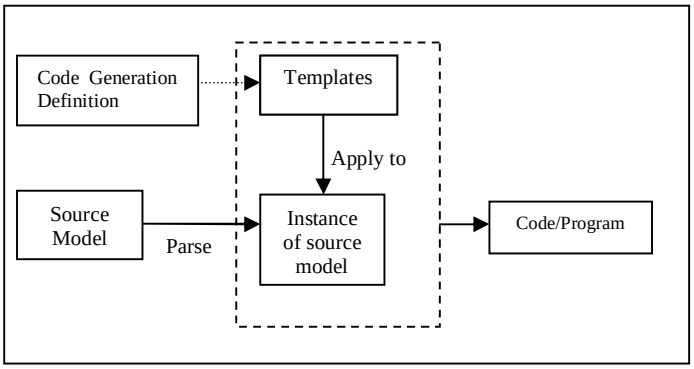

Fig. 2. Templates and Metamodel code generation technique

- **Source model:** The source model is in textual XMI/XML form. It contains the set of all the models designed on different levels of abstractions.

- **Filter:** Source model is filtered to obtain a subset of the source model. This subset contains more summarized and abstracted view of the system which is simple enough for the code to be generated. This filtered source model contains all the details necessary for code generation and the extraneous details are left out.
- **Templates:** There are readymade templates already present for code generation. These templates are instantiated using values of the filtered source model.
- Result yields the code/program

Following are some drawbacks of template and filtering code generation:

1) Templates become very complex for larger examples
2) Approach tightly couples the generation definition (templates) to the concrete syntax of the model
3) It yields low maintainability if source modeling language evolves

B. Templates and Metamodel

Templates and Metamodel is an extension of the templates and filtering pattern. Instead of applying patterns directly to the model, first a metamodel is instantiated from the specification. The templates are the specified in terms of the metamodel. The metamodel can be extended to include domain or architecture specific aspects. Executable code is generated from models. It is based on the metamodel and uses templates to attach to-be-generated source code.

By definition, a metamodel is the model of a model. Hence in the case of templates and metamodel code generation technique, it refers to the model representing the semantics of the input source models. The templates are applied to this metamodel instead of the input source model for generating the code. The instantiated templates produce the final code as output. Fig. 2 shows the generation of code by templates and metamodel technique. The technique is described below:

- **Source model is parsed** in order to create instance of source model meta model
- **Templates** are defined in terms of source model meta model terms
- **Templates are instantiated** using values of the instance of the source model meta model
- Result yields the code/program

C. API based generators

API-based generators provide an API against which code-generating programs are written. This API is typically based on the metamodel/syntax of the target language. Client program uses API which is based on a Grammar. Fig. 3 shows the schematic of this technique of code generation.

D. Inline Code Generation

Inline code generation describes a technique where code generation is done implicitly during interpretation or compilation of a regular, non-generated program, or by means of a precompiler. This process typically modifies the program that is then subsequently compiled or interpreted. Fig. 4 shows the inline code generation technique.

- **Code generation is done implicitly** by means of a precompiler
- The precompiler modifies the program which is then compiled or interpreted
- Examples are common in the programming language domain (C++ precompiler)

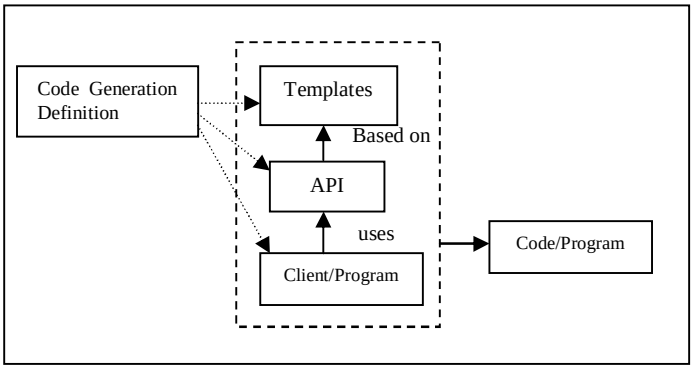

Fig. 3. API based generation technique

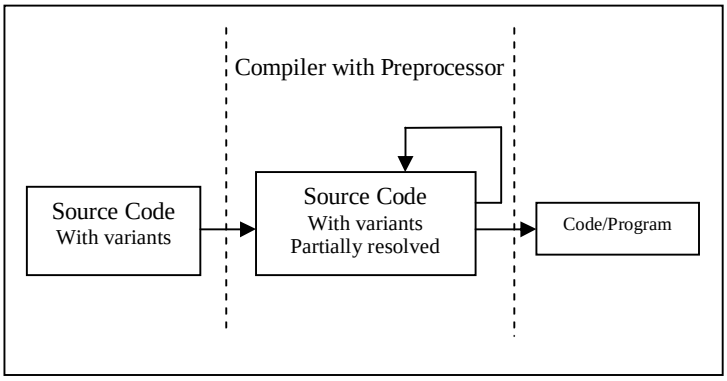

Fig. 3.Inline code generation technique

VI. Comparison of Various Code Generation Techniques

The various approaches for code generation discussed above have their own advantages and disadvantages. One can choose any technique of code generation that suits the requirements and approach. Table 1 presents a brief comparison of the various approaches discussed so far. The serial numbers 1, 2, 3 and 4 correspond to 1)Templates and Filtering, 2)Templates and Metamodels, 3)API-based and 4)Inline code generation respectively.

VII. Merits and Demerits of Model Driven Software Engineering

As every other paradigm, model driven software engineering also suffers from various drawbacks and has many advantages too. The pros and cons of using MDSE depend upon its merits and demerits which are discussed below:

A. *Advantages*

1) Higher productivity: Since tedious and boring parts of the code are generated automatically, productivity is much higher. Code generators produce thousands of lines of code in seconds therefore less time is taken in software development.

2) Agile development: The software development process using MDSE is agile, since any changes that are thought of unexpectedly during or after development are propagated quickly and efficiently.

3) Improved Quality: Bulky handwritten code have inconsistent quality because of the discrepancies and variations in knowledge during development. These discrepancies result as various developers work on different abstract views of the same software. Hence the bug fixes and code improvements resulting due to human incompetency are considerably reduced using a generator, thus improving the overall quality of the produced software.

4) Greater consistency in API design and naming conventions because instead of programming the code in fragments by various coders, it is done by a single generator.

5) More time can be given to the designing part of the software as time is reduced in coding and implementation. This increased design time helps in making the software more efficient and robust.

6) Architectural consistency: The resulting software is consistent with the proposed design as the programmers work within the architecture. Well-documented and maintained code is generated which provides a consistent structure and approach.

7) Abstraction: Abstraction is raised to a higher level using MDSE approach. The software can be defined independent of the language to be used for its coding. Thus it can be ported to different platforms and languages easily.

Table 1. Comparison of various approaches of Code generation [Source: adapted from Voelter]

	Generated/ Ungenerated code	Template / API	Learning Complexity	Suitability for complex uses	Suitability for model-to-code transformation
1	Separate	Template	Simple	Not very good	Good
2	Separate	Template (plus m-model API)	High	Very good	Very good
3	Separate/ integrate	API	Depends on API	Depends on API	Not very good
4	Integrated	Template / API	Simple	Not very good	Not very good

B. *Disadvantages*

1) Code generator has to be written first which is in itself a herculean task and moreover it is not guaranteed that once we write a code generator, it will be sufficient for all the other software projects.

2) The generic approach is not always applicable to all the cases.

3) There will always be some code that has to be hand written pertaining to the peculiarity of the particular domain of the problems.

4) Generality can be a drawback, for instance:
- *a)* Databases must be well-designed and normalized
- *b)* Grammars must be member of a specific class

5) Many companies have invested huge amounts in their existing traditional hard coded software systems and thus are reluctant to adopt a new approach.

CONCLUSION

MDE is the next big thing in software technology. It has industrialized the field of software engineering. Machine made codes have greatly reduced the time and efforts needed to develop software. In fact it has allowed us to make more complex real time software systems which have higher efficiency, high portability and are more readable and maintainable. MDE has not only been successful in reducing the gap between problem specification and implementation, but also made possible the solving of problems in the domain of their definition only rather than converting them into the specific domain of some programming language.

Models are created for the software according to different abstract views based on structure, functions, environment etc and then the required models are combined together and used to generate the code for the software. There are various code generating techniques which can be used based on their merits and demerits and the software to be developed.

This approach of Model Driven Engineering in software development has its own advantages and disadvantages. But the advantages outweigh the disadvantages by huge margins. The only hindrance in adopting MDE on larger level is that many industries have invested large sums of money in their existing software systems and hence they do not want to switch to MDE. Nevertheless this is the next step in software evolution and will be accepted eventually.

ACKNOWLEDGMENT

The author has studied various research works on this topic by different scholars and industry experts. All of the works have been extremely helpful, but the work of Systems and software consortium inc. deserves special acknowledgement. Author would like to thank SSCI for its webinar series. Author also thanks Rajesh Purohit, Professor and head, department of computer science and engineering, M.B.M. Engineering College, for his motivation for writing this paper and his valuable feedback on the draft of paper.

REFERENCES

[1] Douglas C. Schmidt, Model Driven Engineering, Computer, published by IEEE Computer Society(February 2006))

[2] Mark R Blackburn, “What’s model driven engineering (MDE) and how can it impact process, people, tools and productivity”, Systems and software consortium(September 2008).

[3] Dr. Jochen Küster, “Model-driven software engineering- code generation”, IBM Reasearch, Zurich, 2011

[4] Markus Voelter, “Code generation” in Model Driven Software Development, 2006.

[5] Mark van den Brand, “Program generators and model driven architecture”, cbse 2007.

[6] Matthew J, Rutherford, Alexander L. Wolf, “A case for test-code generation in model driven systems”, Technical Report CU-CS-949-03 April 2003, Dept of computer science, University of Colorado.

[7] Lionel Briand, Davide Falessi, Shiva Nejati, Mehrdad Sabetzadeh, Tao Yue, “Research-based innovation: a tale of three projects in model-driven engineering”, MODELS 2012, pp 793-809, Springer-Verlag BerlinHeidelberg(2012)