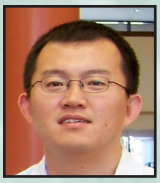
ISR Institute for Software Research
University of California, Irvine

A Rationalization of Confusion, Challenges, and Techniques in Model-Based Software Development



Yongjie Zheng
University of California, Irvine
zhengy@ics.uci.edu

Richard N. Taylor
University of California, Irvine
taylor@ics.uci.edu

August 2011
ISR Technical Report # UCI-ISR-11-5

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

A Rationalization of Confusion, Challenges, and Techniques in Model-Based Software Development

Yongjie Zheng and Richard N. Taylor[1]

Institute for Software Research, University of California, Irvine,
Irvine, California 92697
{zhengy, taylor}@ics.uci.edu

Abstract. The use of model-based software development is increasingly popular due to recent advancements in modeling technology. Numerous approaches exist; this paper seeks to organize and characterize them. In particular, important terminological confusion, challenges, and recurring techniques of model-based software development are identified and rationalized. New perspectives are provided on some fundamental issues, such as the distinctions between model-driven development and architecture-centric development, code generation, and metamodeling. On the basis of this discussion, we opine that architecture-centric development and domain-specific model-driven development are the two most promising branches of model-based software development. Achieving a positive future will require, however, specific advances in software modeling, code generation, and model-code consistency management.

Keywords: Model-Based Software Development, Model-Driven Development, Architecture-Centric Development.

1 Background and Organization

A software model is an abstraction over some aspect of software product. It is most frequently–though not exclusively–used in software development for the purpose of documentation. *Model transformation* [1] is an automated process of taking one or more source models as input and producing one or more target models as output, while following a set of transformation rules. One of its instances, model-implementation "mapping", provides a new application context for software models, wherein system implementations can be directly generated from abstract models. Specifically, it consists of the activities of code generation and consistency management between generated code and source models.

Model-based software development (MBSD) is based on software modeling and model-implementation mapping. It is a paradigm where models are used not only horizontally–to describe and analyze–but also vertically to synthesize, integrate, and

[1] ISR Technical Report # UCI-ISR-11-5, August 2011. This work is supported in part by the National Science Foundation under grants CCF-0917129 and CCF-0820222.

evolve software systems. The primary advantage is that software productivity and quality are improved, given that software models abstract away certain implementation details and are much closer to the problem domain relative to programming languages. There are a number of specific approaches of MBSD, such as application generation, model-driven architecture, and generative software development. They differ in various ways, but all use software models - e.g. domain variations, design models, or system specifications - to create or directly execute a software system.

A classification framework is presented in this paper to enable comparison of varying model-based development approaches along a set of criteria. In particular, important terminological confusion, challenges, and recurring techniques of MBSD are identified and rationalized. New perspectives are provided on some fundamental issues, such as the distinctions between model-driven development and architecture-centric development, code generation, and metamodeling. The goal is to differentiate close concepts, cross-fertilize ideas across different approaches, and outline future research in MBSD.

The paper is organized as follows. Section 2 describes existing model-based development approaches, and presents the comparison framework. Based on the description, Section 3 analyzes and clarifies two confusing issues: model-driven development versus automatic programming, and model-driven development versus architecture-centric development. Section 4 highlights three research challenges in terms of what they are, why they are hard, and how existing methods are deficient in addressing them. Section 5 introduces fundamentals, different usages, and important issues of techniques that recur in MBSD. Finally, Section 6 describes related survey/comparison work, and Section 7 concludes the paper.

2 Model-Based Software Development

This section briefly introduces existing model-based development approaches, and organizes and characterizes them using a classification framework. This introduction serves as a basis for the discussion in following sections.

2.1 Overview

Model-based development approaches can be roughly classified based on the primary abstraction level of their focal software model. We consider four: specification-driven development, model-driven development, architecture-centric development, and generative and component-based development. Fig. 1 provides a simplified illustration of these approaches with important artifacts and code generation processes explicitly represented. Notice that generative and component-based development is included because composition specification can also be seen as a model that emphasizes composition abstractions.

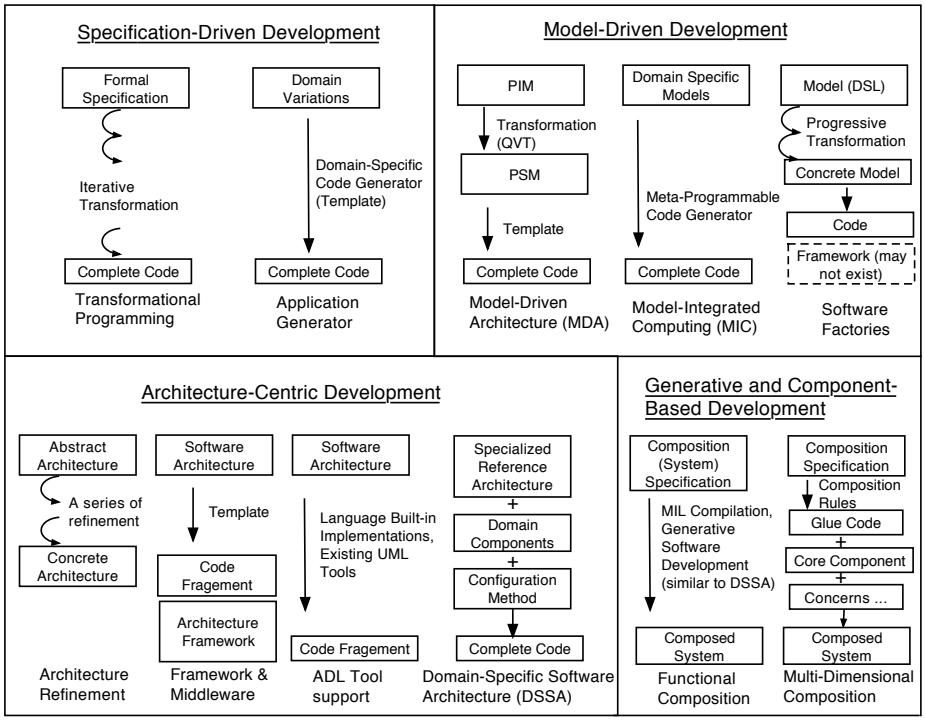


Fig. 1. Model-based software development.

Specification-driven development uses *requirements specifications* to create or directly execute applications. It appears in two forms: transformational programming [2] and application generation [3]. Transformational programming is a methodology of program construction by successive applications of transformation rules. Ostensibly this process starts with a formal statement of a problem, and ends with an executable program. Application generators are tools for creating a family of applications, and are deeply rooted in domain engineering. An application generator translates a highly-particularized specification that expresses variations in a domain into a complete implementation. To change or modify a product, one simply changes input specifications and reruns the generator.

Model-driven development (*MDD*) typically focuses on *software design models* [4]. Initiatives include Model-Driven Architecture (MDA) [5], Model-Integrated Computing (MIC) [6], and Software Factories [7]. MDA was defined by the Object Management Group (OMG) in late 2001, and represents a generic model-driven approach. Specifically, software development in MDA starts with a Platform-Independent Model (PIM) of an application's business functionality and behavior, constructed using Unified Modeling Language (UML) based on OMG's MetaObject Facility (MOF). MDA development tools then convert the PIM to one or more Platform-Specific-Models (PSMs) and finally to a working implementation using some middleware platform. MIC and Software Factories are both domain-specific MBSD. MIC was originally designed for embedded software development. It

advocates the application of different types of models written in domain-specific languages (DSLs), and manages the interdependency between models at the meta-level. The primary goal of Software Factories is to industrialize software development through the integration of abstraction, granularity, and specificity. Progressive refinement is extensively used in it to generate executables from source models.

Architecture-centric software development places an emphasis on the essential role of *software architecture* throughout the software development lifecycle. Software architecture is the set of principal design decisions about a software system [8]. It is commonly characterized as a configuration of components and connectors with enforced constraints, though this is a simplification of the concept. Fig. 1 illustrates four varieties: (a) style-based architecture refinement [9], (b) framework and middleware-based development [10], (c) architecture language support [11], and (d) domain-specific software architectures (DSSA) [12]. Architecture refinement (a) maps an abstract software architecture into a concrete architecture that contains more implementation concerns. Architecture framework techniques (b) essentially raise the abstraction level of an execution platform by providing specific programming constructs for selected architecture concepts. Notice that both architecture refinement (a) and framework techniques (b) are architecture style specific, and thus, could be reused among architectures of the same style. Architecture description languages (ADLs) (c) provide notations for capturing architectural decisions; such languages are usually accompanied by tool support. DSSA (d) represents a combination of software architecture and domain engineering. It consists of a reference architecture, a component library, and a configuration method for combining components.

Models in *generative and component-based approaches* are *composition specifications*, from which glue code is generated to combine existing components into the final artifact. Component composition combines two or more software components and yields new component behavior at a different level of abstraction [13]. Functional composition and multi-dimensional composition are two distinguished composition approaches. The former breaks up a complex software system into smaller components with functional relationships as the primary criterion, while the latter emphasizes separation of overlapping concerns along multiple dimensions of decomposition. A typical example of functional composition is generative software development [14], which focuses on automating the selection and assembly of components. Multi-dimensional composition distinguishes the notion of core components from concerns. An example is Multi-Dimension Separation of Concerns (MDSC) [15].

2.2 Comparison

Table 1 offers a general comparison of the MBSD approaches introduced above along four dimensions: *Goals* reveal the underlying rationale of each MBSD approach; *Software Modeling* shows their focal software models; *Code Generation* represents the form of generated code, which could be full code generation, code fragments, or glue code; *Consistency Management* describes how changes, especially code changes made after code generation is done, are handled.

Table 1. A classification framework of model-based software development.

	Specification-driven development	Model-driven development	Architecture-centric development	Generative and component-based development
Goals	Remove coding phase	Make model compilable and executable	Use software architecture as blueprints	Functional and non-functional reuse
Software Modeling	Requirements specifications	Software design models	Software architecture	Composition specifications
Code Generation	Full code generation	Full code generation	Code fragments and skeletons	Glue code
Consistency Management	Code changes not allowed	Code changes not allowed	Reverse engineering, runtime monitoring, and round-trip engineering	Pending on the preservation of component boundaries

Of these different approaches, specification-driven development is most ambitious in the sense that it tries to transform programming into a specification-based activity so that even requirement engineers can develop software products. MDD and architecture-centric development both focus on software design. In particular, MDD suggests a paradigm where software models take the role of traditional programs, and are the main artifacts of development. This makes it distinguished from architecture-centric development, which recognizes the essential role of both architecture and implementation. Further discussion on this issue is given in Section 3. Finally, generative and component-based development faces a challenge different from the first three approaches, which are all about spanning abstraction gaps. People can decompose a system at their favor, using either functional or multi-dimensional decomposition. The real challenge is how to combine decomposed parts into the final system. Moreover, it is often preferred that component boundaries be preserved during the composition process, so that changes to the final system can be mapped back to original components.

3 Confusion

MBSD, especially the approaches of MDD and architecture-centric development, are still in the early stage of their development. A variety of questions about them persist, due in no small part to each technique using similar terminology but in (sometimes) substantively different ways. What makes MDD essentially different from automatic programming in terms of model-implementation mapping? How is architecture-centric development related to MDD? Clarifications would not only increase people's confidence about related model-based approaches, but also help them make a solid choice when facing a development problem. Specific analysis and clarification of some of these points are provided in this section.

3.1 Model-Driven Development and Automatic Programming

MDD and automatic programming [16] both rely on the machine to generate complete code from software artifacts of a higher-level abstraction. Automatic programming is a special case of transformational programming introduced in Section 2. The problems it is capable of coping with are highly constrained due to the challenge of full code generation. What makes MDD different from automatic programming? If they are essentially same, we could announce, or at least predict, the same result for MDD.

Software development is about making decisions, where pure creativity and automatable activities co-exist [2]. A significant difference between automatic programming and MDD is the role of model-implementation mapping and the creativity required in these two approaches. Automatic programming transforms a high-level specification, usually a formalized requirement specification, into a complete source code. Most decisions, such as the implementation of data structures and optimization of algorithms, are predefined as transformation rules and reused in the development of different systems. The selection and application of transformations are performed by a machine via artificial intelligence techniques. In other words, model-implementation mapping in automatic programming **makes** software development decisions. People's creativity plays a very limited role in this process. This is contrary to MDD, where creativity is particularly emphasized for system designers to consider design tradeoffs. Instead of relying on predefined decisions, system designers make important design decisions of a system, and get them specified in software models. What is automated in MDD is actually the **transfer** of model decisions into implementations, or model-implementation mapping.

The difference becomes obvious if we compare source models and generated code in automatic programming and MDD. Both of them generate complete source code. However, most decisions in generated code of MDD are actually specified by designers in source models, and this is an important reason that MDD emphasizes complete and precise modeling [17]. In contrast, the information difference between source model and generated code of automatic programming is huge, considering that its model is a formal statement of the problem, and talks very little about how to solve the problem. Development decisions are either predefined as transformation rules or automatically made by the machine in transformation selections.

This explains why automatic programming can only be used in development of highly constrained applications, where either a reusable solution scheme prevails or the system is of a limited complexity level. If it is a generic complex system development that is constantly subject to pressures of change, the decision space could be too huge for the machine to predefine or reason about. MDD has potentials to succeed in this scenario, given that the machine focuses on transferring decisions in model-implementation mapping and software designers can concentrate on creative modeling portions.

3.2 Model-Driven Development and Architecture-Centric Development

MDD and architecture-centric development are two categories of MBSD that come into being in recent years. Both of them are software design oriented, could possibly use UML as their modeling languages [5, 8], and face the challenge of dynamics modeling and correctness-preserving code generation. How are MDD and architecture-centric development related with each other then?

As shown in Table 1, MDD is different from architecture-centric development in several aspects. First of all, the rationale behind MDD is to make software design models compilable and executable, so that software developers can solely focus on abstract models. To achieve this goal, software models must have sufficient detail to enable full code generation. In contrast, architecture-centric development uses software architecture as the blueprint where principal design decisions are laid out. Its code generation process is primarily about generating architecture-prescribed code. In most cases, the generated code is some application skeleton that needs software developers to fill in details. From this perspective, the uses of UML in these two approaches are actually in different modes [18]: *UmlAsProgrammingLanguage* in MDD and *UmlAsBlueprint* in architecture-centric development.

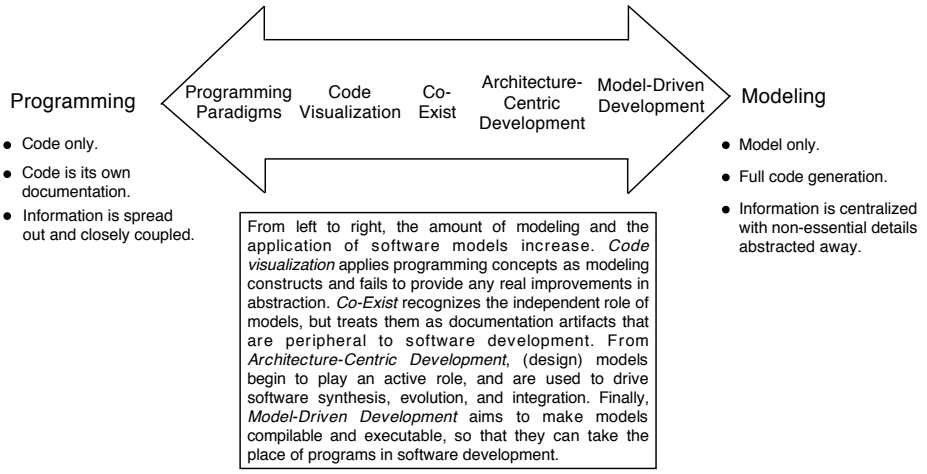


Fig. 2. The programming-and-modeling spectrum.

Fig. 2 sketches the programming-and-modeling spectrum, and is based on discussions in [19, 20] and our own understanding. It compares MDD and architecture-centric development in a more general context: the amount of programming and modeling in software development. “Programming people” think there should be nothing but code, while “modeling people” think models rule everything. Architecture-centric development can be regarded as a transition from one extreme to the other, with models playing an active role in software development. Its position in the middle represents the “right” level of modeling at this point in the evolution of software development technology. We believe it mixes just the right amount of modeling with programming to maximize the effectiveness of both. MDD

is represented as a short range in the figure, reflecting current practice. In addition, *code visualization* and *co-exist* are also shown in the figure as references. They represent different (primitive) usages of software models in traditional software engineering.

Both MDD and architecture-centric development are currently evolving. The fact that the two approaches have so many commonalities suggests a future merger might be possible. For example, MDD can be seen as a subset of architecture-centric development if we consider full code generation as a special case of architecture-prescribed code generation. This is particularly true with software architecture defined as a set of principal design decisions about a software system [8], which essentially include design models that are usually created in MDD approaches. Some form of co-investigation or unification should be able to facilitate the development of both areas.

4 Challenges

What MBSD suggests is essentially a role transition of software models from documentation to development. This implies an enhanced requirement on software models for completeness and precision, compared with the traditional use of models. It also demands an efficient mechanism of model-implementation mapping, which is not only about generating model-prescribed code, but also about managing the consistency between model and code over the passage of time. In general, no MBSD approach can survive in the long run if the cost of model-implementation mapping significantly exceeds that of working on code directly. This section describes three research challenges of MBSD from the perspectives of what they are, why they are hard, and how existing mechanisms are deficient in addressing them.

4.1 Multi-Aspect Modeling

Software models in the development of complex software often need to describe the system from multiple aspects, such as structure, behavior, and non-functional properties. Important research progress has been made in this area [21, 22]. However, most of existing modeling technologies are based on the assumption that software models are documentation artifacts that are peripheral to code development. With regard to structure, models such as UML class diagrams may be fine for use in MBSD. With regard to behavior, few models created with current technologies are amenable to software synthesis in MBSD; the situation with regard to non-functional models is even worse. The challenge is that software models in MBSD not only have to contain enough details to generate relatively complete code, but also need to be, and stay, simpler than the software programs created during this process.

Existing behavioral modeling methods include those that are based on formal notations and those that are more informal, but with a practical bias. None, however, provides an appropriate form for MBSD. Formal behavioral modeling methods include the use of process algebras like CSP and the pi-calculus. Providing a basis for automatic analysis is one of their main purposes. They are seldom appropriate for

software development because of their limited expressiveness. In most cases, developers would rather write code directly. Examples of more informal methods include interaction diagrams, state diagrams, and activity diagrams of UML. Traditionally, these methods are mainly for communication and system comprehension. Their incompleteness properties have decided that they cannot be used alone for behavioral modeling in MDD [5], which emphasizes complete modeling. In cases where only executions of significance are concerned, such as architecture-centric development, practical methods like sequence diagrams may be a good choice after some form of extension [21].

4.2 Code Generation

Similar to structural modeling, structural code generation is well understood and not a particular research issue [23]. MBSD brings a new challenge in this regard, however, which requires structural code, behavioral code, or even non-functional code to be automatically generated from source models. This is hard not only because non-structural modeling in MBSD is not yet mature, as introduced in previous section, but also because system dynamics are involved and many more variations need to be considered compared with static structural code generation.

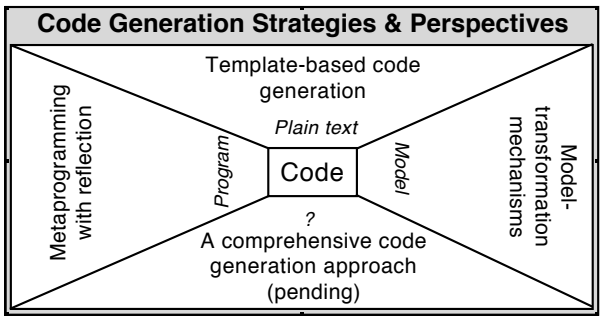


Fig. 3. Different code generation strategies see code differently.

Fig. 3 shows existing code generation approaches and how they treat source code differently. As can be seen, code can be treated as model, program, and plain text respectively. Approaches that treat code as model require the definition of a metamodel for the target programming language, and use model transformation approaches [1] for code generation. It remains to be seen how well these approaches can be practically used in complex software development, especially considering the high complexity that is often involved in model transformation. Approaches that treat code as program are trying to use the target programming language's own metaprogramming ability, e.g. reflection, for code generation. They are limited because they can only be used to generate structural constructs like classes, methods, and attributes.

Treating code as plain text, or template-based code generation [3] represents a popular approach. A typical example is Java Server Pages (JSP) that are used to create web pages, where the Java escapes are executed to produce the dynamic portions of

the HTML page. A primary advantage of the template-based approach is that templates are independent of the target language. This simplifies the generation of any textual artifacts, including documentation. A primary challenge that it faces, however, is verifying the correctness of code embedded in templates that are usually not runnable. Thus, a comprehensive code generation approach that can work as comparably well as a program compiler is still missing. Further development in this area may be pending on a new perspective.

4.3 Model-Code Consistency Management

After code generation is done, chances exist that either the source model has to be modified again or the generated implementation needs additional editing by developers. These changes significantly endanger the conformance established between the model and code. Successful solutions to handle model changes are already available, guaranteeing that extra work done on the generated implementation remains when the system is regenerated. This is usually done through code markers in the form of comments [19], and is not detailed here. In contrast, automatically mapping changes in generated code back to source model is still a research challenge in MBSD. Its difficulty comes from the fact that this is essentially an activity of machine-based abstraction.

Table 2 shows existing mechanisms of model-code conformance management, classifying them along two dimensions. Based on whether inconsistencies are to be avoided or detected, there are approaches of correct-by-construction and correct-by-detection. In general, prevention is always better than cure given that some inconsistencies may be too expensive to be detected and resolved. Approaches of each category can be further divided into one-way mapping and two-way mapping, depending on which artifact can be manually changed. Note that correct-by-detection approaches are usually used to map updated code to model, and assume the relative constancy of model. This explains why there are no two-way mappings of correct-by-detection. Finally, the italicized words in the table represent specific instances of each conformance management approach.

Correct-by-construction approaches are extensively used in MBSD to avoid inconsistency from the very beginning. Among them, one-way mapping approaches try to generate complete code, so that manual modification of code is not a necessity and chances of inconsistency can be reduced. As discussed later in this paper, we believe this can only be done in a domain-specific manner. Two-way mapping approaches in this category include separation of generated and non-generated code, architecture frameworks, and the adoption of new implementation strategies. These can only enforce structural conformance between model and code. A new trend in this area is the use of round-trip engineering [28], where traceability links between model and code are used to automatically propagate updates in derived code back to the model. Initial exploration in this direction shows some promising results. However, further investigations are still needed on some specific issues, such as the granularity of linked objects and the evolution of trace links. In particular, a successful utilization of round-trip engineering in complex software development is still missing.

Table 2. Model-code conformance management.

	Correct-by-construction (to avoid inconsistency)	***Correct-by-detection (to detect inconsistency)***
One-way mapping	1. Full code generation: *MDA* [5], *Domain-specific MDD* [20], *DSSA* [12]. 2. Architecture refinement: *SADL* [9].	1. Reverse engineering: *Reflexion model* [25]. 2. Runtime monitoring & verification: *Pattern-Lint* [26], *DiscoTect* [27].
Two-way mapping	1. Code generation (separation): *EMF* (*code markers*) [19]. 2. Architecture framework: *C2* [10]. 3. Implementation strategy: *ArchJava* [24]. 4. Round-trip engineering: *???*	None.

Correct-by-detection approaches address the conformance issue through after-the-fact consistency checking done either through reverse engineering [29] based static analysis or runtime monitoring verification. Reverse engineering abstracts source models from modified implementations, and compares the original source model with the generated one. It can be expensive for complex systems; moreover, it is hard to guarantee that the generated model captures the same aspects that the original source model contains, since they may represent two different abstractions of the same implementation. Runtime monitoring approaches infer the system architecture from execution traces or system events that are collected at runtime. They are favorable in terms of being able to check the system behaviors against the original architecture. To do this, the availability of executable software is usually required. Some approaches also demand certain forms of code instrumentation. This prevents dynamic verification from being used at development time, when programs are often not complete enough to be executed.

5 Techniques

Several techniques recur in MBSD, including exploitation of domain specificity, metamodeling, and iterative transformation. They represent promising attacks on the challenges identified in Section 4. Introductions exist [2, 5, 20], but are mostly isolated and specific to the particular model-based approach where the technique is applied. Notably, different aspects of a technique are often emphasized in different application contexts. For each of the techniques mentioned above, this section (1) reveals the fundamentals that make the technique promising, (2) presents different usages of the technique, and (3) highlights some challenge issues. Doing so supports cross-fertilization of techniques across different model-based approaches, and encourages wider use in the future.

5.1 Domain Specificity

Fundamentals. Exploiting domain specificity is primarily about developing artifacts that may be reused in developing multiple applications within a given domain. In domain-specific MBSD, reusable assets include DSLs, domain components, and reference architecture. The use of DSLs raises the level of abstraction, and improves the expressive power of software models. A library of reusable components supports software implementation through component composition. Reference architectures serve guides to the composition process. They simplify the management of supplier relationships by describing the specific contexts in which components operate.

Different usages. Domain-specific MBSD includes application generators, MIC, DSSA, and generative software development, all of which were shown in Fig. 1. A significant discriminator of these four approaches is the domain asset being reused. The application generator approach reuses code generators; MIC uses DSLs to model embedded systems; DSSA and generative software development both recognize reference architectures, domain components, and configuration knowledge as reusable assets. DSSA is different from generative software development because the latter uses a configuration generator to implement configuration knowledge and automate the selection of components [14], whereas this is usually done manually in DSSA. In addition, the creation of reference architecture in generative software development is primarily to identify "uses" dependencies between component categories and facilitates the implementation of components. In contrast, the DSSA approach uses reference architectures as a key element in the creation of a specialized architecture.

Issues. The exploitation of domain specificity plays a significant role in MDD, which faces the challenge of complete modeling and full code generation. What a generic MDD (e.g. MDA) does is directly specifying system (dynamic) details in software models. This not only makes models complicated and potentially degrades their usability, but also imposes a high requirement on the extensibility of the modeling language used. Domain-specific MDD [20] is much more favorable at this point. On the one hand, a DSL is more expressive than a generic modeling language (e.g. UML) when applied in a specific domain. One the other hand, reuse of domain-specific code generators or components greatly reduces the amount of generated code, and thus, the information that has to be specified in software models.

5.2 Metamodeling

Fundamentals. A metamodel is a model that is written in a metalanguage to define some specific modeling language [5]. In essence, metamodeling is important because it provides a means for the machine to read, write, and understand models that previously were interpreted only by people. From this perspective, metamodeling plays a key role in automating MBSD. With models understandable to computers, tools can be built for model creation, code generation, and consistency management.

Different usages. Metamodeling is primarily used in MDD and architecture-centric software development. A representative example is MDA, which is based on OMG's four-layer meta-level hierarchy [5]. Its primary modeling language, UML, is defined by a metamodel written in MOF. Different from MDA, MIC as another MDD

approach uses UML as its metalanguage to define its DSLs. In particular, MIC includes a generic modeling environment that can be customized by the metamodel of a domain language to support modeling in a given domain. At this point, it is very similar to ArchStudio [30], a metamodeling based tool for architecture-centric software development. The modeling notation used by ArchStudio is xADL, an XML-based architecture description language. Significantly, users are allowed to extend the schemas of xADL for new features. ArchStudio reads schemas and automatically generates a data-binding library for new tools.

Issues. Meta level and software abstraction level are two different concepts in MBSD. Meta level reflects the linguistic instance-of relationship between a model and its metamodel. In other words, a model is written in a language that is defined by the model's metamodel at a higher meta level. In contrast, software abstraction level characterizes a software model in terms of to what extent it hides unimportant information to a software developer. For example, the abstraction provided by software architecture allows a software architect to focus on principal design decisions without worrying about implementation details. From this perspective, meta level and abstraction level are orthogonal concepts.

5.3 Iterative Transformation

Fundamentals. Iterative transformation is extensively used in transformational programming. The central idea is to break a transformation that crosses an abstraction gap into sufficiently small steps, so that each step generates another representation that is easier to implement than the first. What this means in the context of MBSD is an incremental way to map source models into implementations, especially when source models are too abstract to directly generate code from.

Different usages. Style-based architecture refinement is just a typical application of this idea. It maps an abstract architecture into a concrete architecture through a series of small transformations, each of which involves the application of a pre-proved transformation pattern that is specific to an architecture style. Software Factories, shown in Fig. 1, use a similar approach, so called progressive transformation, to map domain-specific models into implementations. Layers of simplifying abstractions are successively generated during this process. Another less obvious example is MDA, where the use of PSM to facilitate the mapping of PIM to a working implementation on a middleware platform actually reflects the same spirit of iterative transformation.

Issues. The applications of iterative transformation presented above are all limited to certain ranges, such as a specific architecture style, an application domain, or a middleware platform. In addition, their source and generated models usually stay close in terms of conceptual level. At this point, we think this represents proper uses of iterative transformation. Not only the development portion that can be pre-planned and specified is increased, but also the complexity level is reduced. This is different from automatic programming discussed in Section 3, which assumes software development can be pre-planned in a generic way and, in general, faces a significant conceptual gap between requirements specifications and executable programs.

6 Related Work

Papers such as [4, 17, 31] specifically discuss the advantages, difficulties, and facilities of MDD, an important branch of MBSD. This paper, in contrast, studies MDD in a broader context, and compares it with other MBSD approaches. On the basis of this comparison we have reached some different conclusions. For example, [4, 31] both point out that modeling languages is an area that needs to be improved for the advancement of MDD. We think the reason existing modeling languages cannot suffice for MDD is partially because the goal of making software models compilable and executable is overly ambitious. As discussed in Section 5, this may be done in a domain-specific manner, yet we doubt this can be done in a generic MDD. In addition, automatic code generation is described as reaching a degree of maturity in [17]. We think this is only the case when the abstraction gap is fairly small, for example, from low-level implementation models to code. Given that models in MBSD could be a high-level architecture, code generation, especially non-structural code generation, is still a research challenge.

7 Conclusion

Putting all these together, we opine that architecture-centric development and domain-specific MDD are the two most promising branches of MBSD. This is not only because their focal models are good at revealing system essentials and have some desirable properties, such as design orientation, extensibility, and reusability, but also because they are realistic with regard to model-implementation mapping with existing facilities. Achieving a positive future will require, however, specific advances in software modeling, code generation, and consistency management.

References

1. S. Sendall, and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development," IEEE Software, vol. 20, no. 5, 2003.
2. Partsch, H. and Steinbrüggen, R. 1983. Program Transformation Systems. ACM Comput. Surv. 15, 3 (Sep. 1983), 199-236.
3. Cleaveland, C. C. and Cleaveland, J. C. 2001 Program Generators with XML and Java with CD-ROM. Prentice Hall PTR.
4. France, R. and Rumpe, B. 2007. Model-driven Development of Complex Software: A Research Roadmap. In 2007 Future of Software Engineering (May 23 - 25, 2007). IEEE Computer Society, Washington, DC, 37-54.
5. Kleppe, A. G., Warmer, J., and Bast, W. 2003 MDA Explained: the Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc.
6. Sztipanovits, J. and Karsai, G. 1997. Model-Integrated Computing. Computer 30, 4 (Apr. 1997), 110-111.
7. Greenfield, J. and Short, K. 2004. Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools. Wiley, 1st edition.

8. Richard N. Taylor, Nenad Medvidovic, Eric Dashofy, Software Architecture: Foundations, Theory, and Practice. ISBN: 978-0-470-16774-8. John Wiley & Sons, ©2009 736 pages.
9. M. Moriconi, X. Qian, and R. A. Riemenschneider, "Correct Architecture Refinement," IEEE Transactions on Software Engineering, vol. 21, no. 4, pp. 356–372, 1995.
10. Medvidovic, N., Oreizy, P., and Taylor. R. N. 1997. Reuse of off-the-shelf components in C2-style architectures. SIGSOFT Softw. Eng. Notes 22, 3 (May 1997), 190-198.
11. Medvidovic, N. and Taylor, R. N. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Softw. Eng. 26, 1, 70-93.
12. Tracz, W. 1995. DSSA (Domain-Specific Software Architecture): pedagogical example. SIGSOFT Softw. Eng. Notes 20, 3 (Jul. 1995), 49-62.
13. G. T. Heineman, and W. T. Councill, Component-Based Software Engineering: Putting the Pieces Together, Reading, Massachusetts: Addison–Wesley, 2001.
14. Czarnecki, K. and Eisenecker, U. W. 1999. Components and generative programming (invited paper). SIGSOFT Softw. Eng. Notes 24, 6 (Nov. 1999), 2-19.
15. Tarr, P., Ossher, H., Harrison, W., and Sutton, S. M. 1999. N degrees of separation: multi-dimensional separation of concerns. In Proceedings of the 21st international Conference on Software Engineering (Los Angeles, California, United States, May 16 - 22, 1999).
16. Balzer, R. 1985. A 15 Year Perspective on Automatic Programming. IEEE Trans. Softw. Eng. 11, 11 (Nov. 1985), 1257-1268.
17. Selic, B. 2003. The Pragmatics of Model-Driven Development. IEEE Softw. 20, 5 (Sep. 2003), 19-25.
18. M. Fowler, "UmlMode", http://www.martinfowler.com/bliki/UmlMode.html.
19. David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. EMF: Eclipse Modeling Framework (2nd ed.). Addison-Wesley Professional.
20. Kelly, S., Tolvanen, J-P., Domain-Specific Modeling: Enabling Full Code Generation, Wiley-IEEE Society Press, 2008.
21. P. Clements, F. Bachmann, L. Bass et al., Documenting Software Architectures: Views and Beyond: Addison Wesley, 2002.
22. Matinlassi, M., Niemelä, E, Dobrica, L. 2002. Quality-driven architecture design and quality analysis method. A revolutionary initiation approach to a product line architecture. Espoo, VTT Publications.
23. Herrington, J. 2003. Code Generation in Action. Manning Publications.
24. Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: connecting software architecture to implementation. In Proceedings of the 24th International Conference on Software Engineering (ICSE '02). ACM, New York, NY, USA, 187-197.
25. G. C. Murphy, D. Notkin, and K. J. Sullivan. 2001. Software Reflexion Models: Bridging the Gap between Design and Implementation. IEEE Trans. Softw. Eng. 27, 4, 364-380.
26. Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring compliance of a software system with its high-level design models. In Proceedings of the 18th international conference on Software engineering. IEEE Computer Society, 387-396. 1996.
27. Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. 2004. DiscoTect: A System for Discovering Architectures from Running Systems. In Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society.
28. S. Sendall, and J. Küster, "Taming model round-trip engineering," in Proceedings of Workshop on Best Practices for Model-Driven Software Development, Canada, 2004.
29. E. J. Chikofsky, and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, vol. 7, no. 1, pp. 13-17, January/February, 1990.
30. ArchStudio 4: http://www.isr.uci.edu/projects/archstudio/.
31. Hailpern, B. and Tarr, P. 2006. Model-driven development: the good, the bad, and the ugly. IBM Syst. J. 45, 3 (Jul. 2006), 451-461.