4DT903 Individual Report

Reflections on Model-Driven Software Engineering

Samuel Berg (sb224sc)

January 2026

Contents

1 Core Components and Responsibilities in an MDSE System

A Model-Driven Software Engineering (MDSE) system consists of several fundamental components that work together to transform models into executable artifacts. The **metamodel** serves as the foundation, defining the abstract syntax and structural rules that govern valid models [1]. In our automated test case generation project, we created three distinct metamodels: one for user stories (capturing features, scenarios, and acceptance criteria), one for an intermediate test specification model, and one for the target testing framework syntax.

The **model** itself represents a specific instance conforming to its metamodel, capturing domain-specific information at a higher level of abstraction than traditional code. Our project uses models to represent user stories in formats like YAML, XML, or JSON, which conform to the user story metamodel. The **model transformation engine** is responsible for executing transformation rules that map elements from source models to target models. We implemented model-to-model (M2M) transformations to convert user stories into intermediate test specifications, and model-to-text (M2T) transformations to generate concrete test code in Go, Java, or Python.

Finally, the **model editor and validation framework** enables users to create, modify, and validate models against their metamodels. While our project focuses on programmatic model creation and transformation, the Eclipse Modeling Framework (EMF) provides rich editors and validation capabilities that ensure models remain consistent with their metamodels throughout the development lifecycle.

2 Core Benefits of MDSE

MDSE offers significant advantages that address common software engineering challenges. The most prominent benefit is **productivity through automation** [2]. In our test case generation project, developers can automatically produce comprehensive test suites from user stories, eliminating the tedious and error-prone manual translation process. This automation accelerates development cycles and allows teams to focus on higher-value activities like refining requirements and designing test strategies.

Platform independence and portability represent another key advantage. By working at the model level, we can generate test cases for multiple target platforms (JUnit, pytest, Go testing) from a single source model. This separation of concerns between "what to test" (captured in models) and "how to test" (platform-specific syntax) reduces vendor lock-in and simplifies multi-platform support. Changes to target platforms require only updates to M2T transformations, not to the source models or business logic.

MDSE also enhances **maintainability and traceability** [3]. Models explicitly capture the relationships between requirements (user stories) and their corresponding tests, creating a traceable chain from business needs to executable test cases. When requirements change, modifications to the user story model automatically propagate through transformations, ensuring tests remain synchronized with requirements. This explicit traceability reduces technical debt and helps teams maintain comprehensive test coverage as systems evolve.

3 Challenges and Limitations of the MDSE Approach

Despite its benefits, MDSE introduces several challenges that teams must address. The primary challenge is the **initial complexity and learning curve** [2]. Developers must learn metamodeling concepts, transformation languages (such as ATL or Acceleo), and modeling frameworks like EMF. In our project, creating well-designed metamodels and robust transformations required significant upfront investment in understanding MDE principles and tooling. This learning curve can slow initial development and may discourage adoption in organizations without dedicated modeling expertise.

Tooling maturity and ecosystem fragmentation present practical obstacles. While frameworks like EMF provide solid foundations, debugging transformation logic remains more difficult than debugging traditional code. Stack traces in transformation engines can be cryptic, and there are fewer learning resources and community support compared to mainstream programming. Our project occasionally encountered challenges with Eclipse modeling tools that required workarounds not typical in conventional software development.

The **abstraction gap and loss of control** can also be problematic [4]. M2T transformations generate code that developers may need to customize for specific edge cases or performance optimizations. However, manual modifications to generated code break the model-driven workflow, as regeneration overwrites changes. In our test generation project, handling complex test scenarios with special setup requirements or custom assertions required carefully designed extension points in the metamodel and transformations. Additionally, MDSE may not be cost-effective for small projects where the overhead of creating metamodels and transformations exceeds the benefits of automation.

4 Approaching Problems with MDSE Design

When designing an MDSE solution, I follow a structured approach that begins with **domain analysis and metamodel design**. The first step is to thoroughly understand the problem domain and identify the key concepts, relationships, and constraints. For our test generation project, I analyzed user story structures, acceptance criteria formats, and testing framework requirements. This analysis informed the creation of metamodels that capture essential domain concepts while remaining independent of implementation details. Good metamodel design balances expressiveness (capturing all necessary information) with simplicity (avoiding unnecessary complexity).

The second phase involves **transformation design and implementation**. I identify the mapping between source and target metamodels, considering both structural mappings (how elements correspond) and behavioral transformations (how relationships and constraints translate). For our project, this meant defining rules to transform user story features into test suites, scenarios into test cases, and acceptance criteria into assertions. I typically implement M2M transformations first to create intermediate models that bridge conceptual gaps between source and target domains, then implement M2T transformations to generate concrete artifacts. This layered approach separates concerns and makes transformations more maintainable.

Finally, I emphasize **iterative refinement and validation** [1]. MDSE solutions rarely work perfectly on the first attempt. I start with simple examples, validate the generated output, and progressively handle more complex cases. For each iteration, I verify that generated artifacts are correct, complete, and follow target platform conventions. I also maintain test cases for the transformations themselves, ensuring that changes to transformation logic don't break existing functionality. This iterative approach, combined with continuous validation, helps build robust MDSE solutions that handle real-world complexity.

5 Three Software Engineering Practices That Must Adapt to MDSE

5.1 Version Control and Configuration Management

Traditional version control focuses on source code, but MDSE requires tracking models, metamodels, and transformation definitions [5]. In our project, we version control not just the generated test code, but also the user story models (YAML/XML/JSON files), Ecore metamodel files, and transformation scripts. Teams must establish conventions for what artifacts to commit: typically models and transformations are versioned, while generated code may be treated as build artifacts. Merge conflicts in graphical models can be particularly challenging, requiring model-aware diff and merge tools. Additionally, configuration management must handle dependencies between metamodel versions and the models that conform to them, ensuring compatibility across team members and build environments.

5.2 Testing and Quality Assurance

Testing strategies must expand beyond validating generated code to include testing the MDSE infrastructure itself [6]. Our project required three levels of testing: (1) metamodel validation ensuring models conform to structural rules, (2) transformation testing verifying that transformations produce correct output for given input models, and (3) generated artifact testing confirming that generated test cases actually work on target platforms. Teams need frameworks for model-based testing and must maintain test models representing diverse scenarios. Code coverage metrics also need adaptation—coverage of transformation rules becomes as important as coverage of generated code. Quality assurance must verify both the correctness of individual transformations and the end-to-end pipeline from source models to deployed artifacts.

5.3 Continuous Integration and Build Processes

CI/CD pipelines must orchestrate model transformations alongside traditional compilation and testing [7]. In our project, the build process involves: (1) validating input models against metamodels, (2) executing M2M transformations, (3) executing M2T transformations to generate code, (4) compiling generated code, and (5) running generated tests. Build scripts must handle metamodel evolution, regenerating code when metamodels or transformations change. Teams need to decide whether to commit generated artifacts or regenerate them in each build—the former improves build speed and allows reviewing generated code changes, while the latter ensures consistency with current transformations. Build failures can occur at multiple stages (model validation, transformation execution, or generated code compilation), requiring more sophisticated error handling and reporting than traditional builds.

6 Points Distribution

As this was an individual project completed solely by me (Samuel Berg, sb224sc), I assign all 10 points to myself:

- Samuel Berg (sb224sc): 10 points

References

[1] M. Brambilla, J. Cabot, and M. Wondrash, *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

[2] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 633–642.

[3] T. Stahl and M. Völter, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[4] P. Mohagheghi and V. Dehlen, "Where is the proof? - A review of experiences from applying MDE in industry," in *European Conference on Model Driven Architecture-Foundations and Applications*, 2008, pp. 432–443.

[5] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige, "Different models for model matching: An analysis of approaches to support model differencing," in *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, 2009, pp. 1–6.

[6] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann, "Formal specification and testing of model transformations," in *Formal Methods for Model-Driven Engineering*, 2012, pp. 399–437.

[7] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Science of Computer Programming*, vol. 89, pp. 144–161, 2014.