LLMs as Code Generators for Model-Driven Development

Yoonsik Cheon
Department of Computer Science, The University of Texas at El Paso, El Paso, Texas, U.S.A.

Keywords: Model-Driven Development, Large Language Model, UML/OCL, Code Generation, Dart/Flutter.

Abstract: Model-Driven Development (MDD) aims to automate code generation from high-level models but traditionally relies on platform-specific tools. This study explores the feasibility of using large language models (LLMs) for MDD by evaluating ChatGPT's ability to generate Dart/Flutter code from UML class diagrams and OCL constraints. Our findings show that LLM-generated code accurately implements OCL constraints, automates repetitive scaffolding, and maintains structural consistency with human-written code. However, challenges remain in verifying correctness, optimizing performance, and improving modular abstraction. While LLMs show strong potential to reduce development effort and better enforce model constraints, further work is needed to strengthen verification, boost efficiency, and enhance contextual awareness for broader adoption in MDD workflows.

1 INTRODUCTION

Model-Driven Development (MDD) responds to the growing complexity of software engineering by emphasizing high-level, abstract models instead of manual coding (Stahl and Volter, 2006). A leading approach is the Object Management Group's Model-Driven Architecture (MDA) (Meservy and Fenstermacher, 2005), which attempts to automate transformations from computation-independent models to platform-specific implementations. Central to MDA are standardized modeling languages such as the Unified Modeling Language (UML)for structural design and the Object Constraint Language (OCL) (Warmer and Kleppe, 2003) for specifying precise constraints. The success of MDD depends not only on the expressiveness of these models but also on the availability of tools capable of reliably transforming them into working systems. This paper explores whether combining UML and OCL with large language models (LLMs) offers a practical alternative to traditional platform-specific transformation and code generation tools.

While UML and OCL provide a strong modeling foundation, the automatic generation of executable code from these models continues to pose substantial challenges (France et al., 2006). Realizing the MDD vision of seamless model-to-code transformation has proven elusive. UML often omits implementation-level details, and OCL, though formal and expressive, may not fully capture dynamic behaviors and interactions. Additionally, developing and maintaining platform-specific code generation tools is labor-intensive and error-prone, as these tools must constantly be updated to accommodate new programming languages, frameworks, and execution environments. Traditional generators rely heavily on rigid templates and predefined transformation rules, requiring manual tuning to resolve ambiguities—ultimately limiting their scalability, flexibility, and adaptability.

Recent advances in LLMs offer a promising new direction. With their capability to process both structured inputs (such as models) and unstructured language, LLMs are well-positioned to bridge the gap between high-level abstractions and concrete implementations. Unlike traditional tools, LLMs can adapt to context, generate platform-specific code dynamically, and support natural-language-based refinements and prototyping. Their general-purpose design also suggests the potential to serve as a universal code generation tool, eliminating the need for separate generators tailored to each platform. By automating routine coding tasks and adapting to evolving ecosystems, LLMs could significantly streamline MDD workflows and reduce development effort.

In this paper, we explore the feasibility of using LLMs as code generators within MDD processes. Specifically, we evaluate whether UML and OCL models can be used in combination with LLMs to generate accurate, platform-specific code. To ground our investigation, we present a case study involving

Cheon, Y.
LLMs as Code Generators for Model-Driven Development.
DOI: 10.5220/0013580300003964
In *Proceedings of the 20th International Conference on Software Technologies (ICSOFT 2025)*, pages 386-393
ISBN: 978-989-758-757-3; ISSN: 2184-2833

the development of a Sudoku game app using Dart and Flutter (Bracha, 2016; Flutter, 2025). The study assesses (a) the suitability of UML and OCL for modeling, (b) the ability of LLMs to interpret and translate these models into code, and (c) the practical strengths and limitations of this approach. While focused on a single application domain, our findings offer early insights into the potential of LLMs for generalizing MDD workflows. We also highlight the challenges that remain and identify avenues for future research involving broader and more diverse applications.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 outlines our modeling approach and LLM integration. Section 4 presents a case study involving a Sudoku game, detailing the model design and the LLM-driven code generation process. Section 5 evaluates the quality and correctness of the generated code, and Section 6 summarizes our findings and future directions.

2 RELATED WORK

Traditional MDD code generation has used template- and rule-based approaches (Sebastian et al., 2020). Tools like Eclipse Acceleo use predefined templates to transform high-level models into code (Syriani et al., 2018; Jurgelaitis et al., 2021; Tsiounis and Kontogiannis, 2023), while rule-based methods, including XSLT transformations, convert models using explicit rules (Carnevali et al., 2009; Frantz and Nowostawski, 2016). Although effective, these techniques require significant manual effort to maintain templates and rules, limiting adaptability to changing requirements and domains.

In contrast, LLMs have been widely applied in software engineering tasks such as code generation, refactoring, and providing intelligent assistance (Fan et al., 2023; Hou et al., 2024; Wang et al., 2024; Zheng et al., 2025). Models like OpenAI's Codex support snippet generation, completion, and error detection (Jain et al., 2022; Xue et al., 2024). Research has explored enhancing LLMs via prompt engineering, domain tuning, few-shot prompting, and chain-of-thought reasoning.

However, the use of LLMs in the MDD context remains largely unexplored. Specifically, their potential to generate platform-specific implementations from structured inputs such as UML and OCL has not been thoroughly examined. This study addresses this gap by evaluating LLMs as an alternative to conventional code generation in MDD workflows.

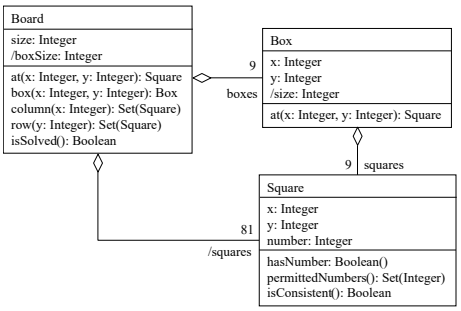

Figure 1: Sudoku class diagram.

3 APPROACH

This study explores the use of LLMs for MDD, transforming platform-independent and platform-specific models into executable code. Our goal is not only to automate code generation but also to evaluate the effectiveness of LLMs in handling well-defined, structured models. Unlike traditional MDD workflows that emphasize transformation toolchains, we focus on assessing how LLMs process pre-defined UML and OCL models. Our model consists of both structural and behavioral components:

- UML for structure: UML diagrams define key elements such as classes, attributes, associations, and multiplicities to model system relationships.
- OCL for behavior: OCL constraints refine UML by specifying invariants, preconditions, postconditions, and derived elements, ensuring consistency and eliminating ambiguities.

This study primarily focuses on UML class diagrams, as shown in Figure 1, which models the Sudoku puzzle (detailed in Section 4). The diagram defines key classes—Board, Box, and Square—and a derived association, squares, linking Board to Square. However, a UML diagram alone does not specify how derived associations are computed or how behaviors are enforced. OCL complements UML by expressing constraints and behaviors that graphical models cannot capture. It defines class invariants, derived associations, and operational constraints to ensure model consistency and preciseness (see examples below).

context Board
inv: size >= 9 and Set{1..size}→exists(i | i * i = size)

context Board::squares: Set(Square)
derive: boxes.squares→asSet()

context Board::at(x: Integer, y: Integer): Square
pre: 0 <= x and x < size and 0 <= y and y < size
post: result = squares@pre→any(s | s.x = x and s.y = y)

These constraints ensure correctness, such as enforcing board size, defining derived associations, and constraining method behavior. The first constraint enforces a class invariant ensuring the board size is a square number of at least 9. The second constraint uses the `derive` clause to compute a derived association. The third constraint defines the behavior of the `Board::at()` operation with a precondition validating index bounds and a postcondition ensuring the correct `Square` is returned, using the pseudo-variable `result` and the `@pre` suffix for pre-state evaluation. As shown, OCL provides powerful collection operations like `exists` and `any` for expressing complex rules (Object Management Group, 2014).

Once the model is defined, it is converted into a structured format for LLM processing, involving: (a) model translation, (b) LLM configuration and execution, and (c) output analysis. UML and OCL elements are encoded as textual data. Structural elements are described hierarchically, while OCL constraints are presented as declarative, textual rules. This study uses PlantUML (PlantUML, 2025) due to its compact, human-readable syntax and wide tool support. For code generation, we use an interactive LLM, such as ChatGPT-4, to iteratively refine outputs. Default configurations are used, except for tailored prompts to guide LLM behavior. The generated code is reviewed for correctness, completeness, and adherence to the model. Results are compared to manually written implementations to evaluate effectiveness.

4 CASE STUDY: SUDOKU APP

To demonstrate our approach, we developed a Sudoku puzzle app (see Figure 2). Sudoku features a 9×9 grid where players fill in numbers 1–9 such that each row, column, and 3×3 sub-grid contains all digits without repetition. Games begin with a partially filled grid—typically with at least 17 given numbers—while players deduce the rest. Fixed values are shown as gray squares in Figure 2. We followed a lightweight model-driven approach using UML and OCL to capture both structure and behavior. This model serves as the basis for automated code generation, ensuring design consistency and precision.

4.1 Modeling with UML and OCL

The first step in our lightweight MDD approach involves constructing a platform-independent model (PIM) using UML and OCL, in line with established MDD practices (Cheon and Barua, 2018). As shown in Section 3, the UML class diagram defines the structural backbone of the Sudoku game, capturing core components—Board, Box, and Square—and their associations. The Board class models the entire 9×9 grid, which is subdivided into Box instances, each containing a 3×3 group of Square objects. A derived association, `squares`, links the Board directly to all its Square instances, enabling holistic analysis and reasoning over the grid.

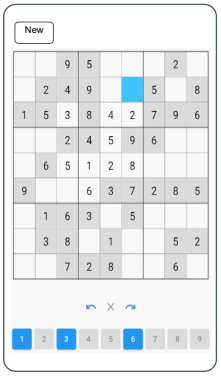

Figure 2: Sudoku puzzle app.

To complement the structural model, we incorporate OCL constraints that capture domain-specific rules and behavioral logic beyond what UML alone can express. These include: (a) class invariants that enforce structural properties (e.g., grid dimensions), (b) operation contracts specifying preconditions and postconditions, and (c) derived attributes and associations formalizing relationships—such as the indirect link between Board and Square via Box. Below are two more example constraints: `isSolved()` in the Board class verifies that all squares are correctly filled, and `permittedNumbers()` in the Square class computes valid entries for a square. To improve readability of complex expressions, we use a custom `where` clause.

context Board::isSolved(): Boolean
body: squares→forAll(hasNumber() and isConsistent())

context Square::permittedNumbers(): Set(Integer)
body: result = all − b − h − v **where**
all = Sequence{1..board.size}→asSet(),
b = box.squares→excluding(self)→collect(number),
h = board.row(x)→excluding(self)→collect(number),
v = board.column(y)→excluding(self)→collect(number)

Together, UML and OCL yield a complete and precise model of the game's logic while remaining platform-agnostic. Our model includes 196 lines of OCL constraints covering structural invariants, derived properties, and behavioral specifications (Cheon and Barua, 2018).

After finalizing the PIM, we develop a platform-

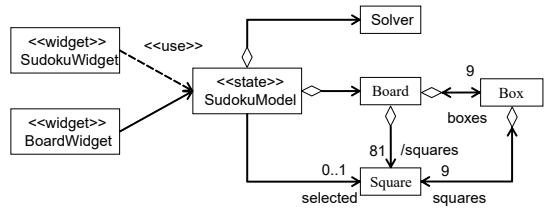

Figure 3: Refined class diagram in PSM.

specific model (PSM) for Dart/Flutter (Bracha, 2016; Flutter, 2025), a cross-platform mobile framework for Android and iOS (Cheon, 2021; Cheon et al., 2023). Flutter's reactive UI model facilitates state management by automatically updating interface components in response to changes in application state. The PSM extends the PIM by incorporating user interface (UI) components and interaction logic while preserving the domain model and its associated constraints (see Figure 3). As illustrated in Figure 2, the UI consists of a 2D Sudoku grid and control buttons, with user interactions such as taps bound to operations defined in the PSM. This integration tightly couples the interface with the game logic, ensuring seamless interaction.

To support this behavior, the PSM introduces several new classes: (a) widget classes to render the Sudoku grid and control buttons, (b) a state management class (SudokuModel) that maintains game state, handles user events, triggers UI updates, and manages undo/redo via helper classes, and (c) a solver class using a backtracking algorithm, aided by additional helper classes not shown in the diagrams. Core logic classes like Board, Box, and Square are reused from the PIM but extended with platform-specific features such as property accessors and new methods. To improve modularity, some operations are refactored into helper classes while preserving key behaviors like move validation and board state management.

The PSM includes approximately 450 lines of OCL constraints: 199 for core logic, 95 for the solver, and 156 for state management. We deliberately exclude UI-specific constraints to focus on ensuring domain consistency and correct state behavior. A central role of SudokuModel is to trigger UI updates when the application state changes. While such behavior can be described using pre/postconditions, these are often too vague to guarantee that specific update methods are invoked. To address this limitation, we use the ^ (hasSent) operator in OCL to assert that particular signals—such as UI notifications—are sent. For example, the `selectSquare()` operation not only sets the selected square but also verifies that `notifyListeners`, the method responsible for updating the UI, is triggered:

context SudokuModel::selectSquare(square: Square)
pre: square <> **null**
post: self.selectedSquare = square
post: self.notifyListeners^

This ensures that user interface elements (e.g., number buttons) are updated in response to the new selection. Similarly, tapping the 'New' button invokes `startNewGame()`, whose postconditions enforce a fresh puzzle, reset selection, cleared undo/redo stacks, and the triggering of `notifyListeners`:

context SudokuModel::startNewGame()
post: informally("board contains a new puzzle")
post: self.selectedSquare = **null**
post: self.history.undoStack→isEmpty() and
self.history.redoStack→isEmpty()
post: self.notifyLiseners^

To capture intent that is difficult to formalize precisely, we introduce an `informally()` clause (see the above example). It communicates developer intent without requiring full formal specification. This hybrid approach balances formal rigor with practical expressiveness, allowing specifications to remain both precise and accessible. While a complete formalization is theoretically possible, it often introduces complexity that can obscure intent—highlighting the value of combining formal and informal constraints.

Finally, to evaluate the reasoning capabilities of LLMs, we include partial and underspecified constraints. For example, the `undo()` operation describes only the change in undo stack size, omitting constraints on the stack's content or the redo stack:

context SudokuModel::undo()
pre: hasUndo
post: history.undoStack→size() =
history.undoStack@pre→size() − 1

These partial constraints serve as a testbed for assessing whether the LLM can infer consistent and intended behavior from incomplete information, challenging it to maintain behavioral coherence in the presence of gaps in formal specification.

4.2 Code Generation and Integration

Dart/Flutter code will be incrementally generated by an LLM and organized into three Dart libraries: (a) core model classes defining domain concepts and relationships, (b) solver classes implementing puzzle solving and generation logic, and (c) state management classes handling game state and UI updates.

This phased approach facilitates systematic verification and smooth integration with existing components. We currently have a fully functional, manually developed app (Cheon, 2021; Cheon et al., 2023). Our goal is to replace all modules except the UI with LLM-generated components while preserving full functionality.

We begin with generating the core model classes (e.g., Board, Box, Square), which form the application's foundation and have minimal dependencies. Next, we generate the solver classes, which add complexity through puzzle-solving and validation logic. Each component is verified against the original design and constraints. Finally, we generate and integrate the state management classes, which involve the most interactions—managing state transitions and triggering UI updates. This iterative, modular strategy supports efficient debugging, refinement, and integration. By starting with the least dependent module, we reduce cascading errors and establish a stable base for subsequent phases.

5 EVALUATION

We evaluated ChatGPT-4 (January 2025, GPT-4o) by prompting it to generate Dart code from UML class diagrams written in PlantUML along with associated OCL constraints. We began with a simple prompt: "I will provide a UML class diagram written in PlantUML syntax and a set of OCL constraints defining additional design requirements. Write a Dart implementation that conforms to this design." Despite the prompt's simplicity, the LLM produced code that closely adhered to both the structural and behavioral specifications. Moreover, when instructed to incorporate a specific state management framework, it correctly imported the relevant package and invoked the appropriate framework methods, demonstrating a strong ability to adapt to additional context.

Three Dart libraries—model, solver, and state management—were generated and integrated into an existing Sudoku app by replacing manually written modules (Cheon, 2021; Cheon et al., 2023). Well-defined interfaces enabled smooth integration. Minor issues during model integration were quickly resolved through small diagram updates and code regeneration; an OCL constraint error was also detected.

The LLM translated OCL constraints into Dart using intuitive mappings (see Table 1). For example, the `Board` class correctly applied invariants, preconditions, and derived associations (see Section 4.1):

Table 1: OCL-to-Dart mapping.

OCL Concept	Dart Implementation
invariant	pre-state assert statement
precondition	pre-state assert for input validation
postcondition	method logic
derived	field or getter
body and def	query method

```
Board([this.size = 9])
  : assert(size >= 9),
    assert(sqrt(size).floor()
      * sqrt(size).floor() == size),
    boxSize = sqrt(size).floor() { ... }

Square at(int x, int y) {
  assert(x >= 0 && x < size && y >= 0
    && y < size, 'Index out of bounds.');
  return squares.firstWhere(
    (s) => s.x == x && s.y == y);
}

Set<Square> get squares =>
  boxes.expand((box) => box.squares).toSet();
```

It made good use of Dart features such as final fields, getters, and optional parameters. Named parameters in Dart were used in place of OCL defaults, while positional optional parameters required explicit square brackets in the UML diagram. Notably, the LLM successfully interpreted custom notation like `where` and `informally()` clauses without special instructions. It also demonstrated inference capabilities. For example, when the `undo()` operation was underspecified—constraining only the stack size—it generated a complete implementation, showing its ability to infer plausible behavior even with partial specifications:

```
void undo() {
  assert(hasUndo);
  _history.undo();
  notifyListeners();
}
```

When some parts of the design were missing, e.g., constructors or dependent classes, the LLM autonomously generated them, fully implemented when sufficient context was available, or as placeholders otherwise. More consistent results could be achieved by explicitly specifying module dependencies and interfaces, which the LLM can also document.

Although generally accurate and functional, the LLM-generated code had a few limitations: (a) it avoided introducing helper methods or abstractions, likely adhering closely to the explicit design, (b) complex solving techniques like backtracking required explicit prompting, and (c) derived attributes (e.g.,

`squares`, `rows`) were recomputed on each access. While this favored clarity, caching could improve performance.

Table 2: Code size and complexity.

Lib	LLM-generated			Hand-written		
	Cl	Meth	SLOC	Cl	Meth	SLOC
Model	3	34 (15)	210	3	37	208
Solver	4	21 (2)	171	4	22	218
State	3	23 (8)	136	3	23	149
Total	11	78 (25)	517	11	82	575

5.1 Code Size

We evaluated the size and structural complexity of LLM-generated code by comparing it to a manually written baseline implementation (Cheon, 2021; Cheon et al., 2023). The generated code comprises 11 classes, 78 methods, and 517 source lines of code (SLOC), excluding documentation comments. Of the 78 methods, 32% (25 methods) are simple getters or setters. Table 2 summarizes the size comparison across the model, solver, and state management libraries.

The LLM-generated code is 10% more compact than the manual version (517 vs. 575 SLOC), even though it includes 31 assert statements—each one or two lines—derived from invariants and preconditions. This suggests that LLMs reduce verbosity without sacrificing structure, producing a similar number of classes and methods. Among the three libraries, the model layer has the most methods (34), reflecting its central role in representing application logic and data. The state management layer follows with 23 methods, while the solver contains 21, aligning with their respective responsibilities. Overall, the generated code matches the structural complexity of hand-written code while offering modest size reductions. Automating boilerplate code—such as getters and setters—may improve productivity. However, further study is needed to evaluate impacts on readability, maintainability, and performance.

5.2 Code Quality

We now compare LLM-generated and handwritten code across key aspects such as structure, performance, error handling, and usability. The goal is to highlight the strengths and trade-offs of LLM-generated code compared to manually written implementations. The analysis is organized into distinct categories, emphasizing the impact of each approach on development efficiency, maintainability, and optimization. A summary of our findings is provided in Table 3.

LLM-generated code is more verbose, with detailed comments, explicit types, and comprehensive error messages. Methods include assertions for input validation, enhancing readability and robustness, making it accessible to beginners. However, this verbosity can reduce maintainability for experienced developers who prefer concise solutions. Handwritten code, on the other hand, is concise, minimizing commentary and leveraging Dart's type inference for efficiency. It focuses on performance, using techniques like caching and optimized property access. While it may present a steeper learning curve due to reduced documentation, it offers better long-term maintainability through modularity and simplicity.

Both codebases include consistency checks and validation, but differ in granularity and modularity. The LLM-generated code provides detailed checks, such as the `isConsistent()` method in the Board class, which verifies constraints for rows, columns, and blocks. This approach improves traceability but can lead to verbosity and code bloat. The handwritten code, in contrast, abstracts validation into private helper methods, reducing repetition and enhancing maintainability. While this modular approach simplifies the code, it relies on external validation or correct inputs, potentially increasing the risk of errors when validation is insufficient.

The data structures used in the LLM-generated and handwritten code strike a balance between simplicity and efficiency. The LLM-generated code uses basic Dart collections like Set and List to store distinct and ordered values, aligning with OCL constraints. However, it lacks caching, leading to repeated recomputation of attributes like rows and columns, which impacts performance. In contrast, the handwritten code uses Iterable and List with caching for derived attributes, improving data structure flexibility and performance. Lazy initialization is employed, computing rows and columns once and storing them for future access, reducing computational overhead. Algorithmically, the LLM-generated code uses a simple deterministic solver that is easy to read but less efficient for complex puzzles. The handwritten code employs a backtracking algorithm with randomness, making it more efficient for handling complex puzzles but introducing additional complexity that may increase the learning curve.

The error-handling strategies in both codebases vary in terms of robustness and usability. The LLM-generated code uses defensive programming, relying heavily on assertions to enforce preconditions and class invariants. These checks help catch potential issues early in development, though they can lead to verbosity. However, assertions are stripped in pro-

Table 3: Qualitative comparison of LLM-generated vs. handwritten code across key software attributes.

Category	LLM Code	Handwritten Code	Observations
Code Style	Verbose, with comments, explicit types, assertions, and direct property modifications.	Concise, modular, optimized for experienced developers, using helper methods.	LLM code is beginner-friendly and robust; handwritten code is efficient and modular.
Validation	Uses assertions for preconditions and explicit validation.	Modular validation with helper methods and defensive checks.	LLM emphasizes robustness; handwritten code prioritizes reusability.
Data Structures	Uses `Set`, recomputes rows/columns without caching, and generates puzzles deterministically.	Uses `Iterable` with cached data, a backtracking solver, and randomized puzzle generation.	LLM favors built-in collections and deterministic behavior; handwritten code is optimized for adaptability.
Error Handling	Defensive programming with assertions and detailed error messages.	Assumes valid inputs; minimal explicit error handling.	LLM code is resilient; handwritten code relies on external validation.
Performance	Recomputes data frequently, lacks caching, and provides fine-grained methods.	Implements caching, reduces redundancy, and optimizes computations.	LLM code is flexible but less efficient; handwritten code is optimized for performance.

duction builds, reducing runtime overhead. Developers must ensure meaningful error handling for production. In contrast, the handwritten code minimizes assertions, relying on simpler checks and external validation. This reduces performance overhead but assumes inputs are validated externally, increasing the risk of errors if invalid data is passed. Without built-in safeguards, the responsibility for correctness falls on developers. The LLM-generated code is more robust and accessible, suitable for beginner-friendly environments, while the handwritten code prioritizes efficiency and simplicity for experienced developers.

The LLM-generated code and handwritten code differ considerably in terms of performance optimization and granularity. The LLM-generated code favors simplicity and flexibility, often recomputing data rather than using caching. This makes the code easier to understand but less efficient, especially in scenarios requiring frequent recalculations. The fine-grained methods offer extensibility but can lead to performance inefficiencies. In contrast, the handwritten code prioritizes performance with caching and consolidated operations, reducing redundant tasks and computational overhead. While this improves efficiency, it sacrifices flexibility, making the code less adaptable to frequent changes.

5.3 Lessons Learned

This case study highlights LLMs' adaptability in translating formal design specifications into functional Dart/Flutter code. LLMs effectively process OCL constraints and handle complex logic, showcasing their potential in model-driven development. Their ability to balance formality by omitting less critical components, like getters and setters, reduces specification overhead and allows developers to focus on high-level design. LLMs can also interpret domain-specific notations, enabling tailored designs and accelerating prototyping, which is beneficial for rapid experimentation.

While LLMs successfully handled complex constraints, including derived attributes and a Sudoku solver, challenges remain in verification, integration, and optimization. Subtle errors can emerge during system testing, emphasizing the need for automated verification. LLMs prioritize clarity over efficiency, often lacking caching and optimization unless explicitly instructed, and their lack of persistent context across interactions requires manual updates to maintain consistency.

The effectiveness of LLM-generated code relies on clear, precise prompts, as ambiguity can result in incorrect implementations or missing constraints. Despite these challenges, LLMs significantly enhance productivity in model-driven development when used with careful oversight, rigorous verification, and iterative refinement.

The choice between LLM-generated and handwritten code depends on the goals of projects. LLM-generated code, with its clarity and beginner-friendly structure, suits collaborative or educational projects but could benefit from performance improvements like caching and disabling assertions in production. Handwritten code excels in performance-critical applications due to its modularity and caching, though adding basic verification could improve robustness without compromising efficiency. LLMs provide versatility in code generation, enabling customization to suit specific project requirements. For example, a straightforward prompt can direct the LLM to generate a backtracking-based solving algorithm, demonstrating its flexibility for performance-focused development.

6 CONCLUSION

This study demonstrates the potential of large language models (LLMs) as code generators in Model-Driven Development (MDD), translating UML and OCL specifications into executable code and enhancing productivity through rapid prototyping. An empirical comparison with manually written implementations showed that LLMs can interpret incomplete or informal designs, handle custom OCL extensions, and respond effectively to prompt engineering. While the well-known Sudoku domain may bias results possibly due to its presence in training data, the study highlights the LLM's ability to generate structured, semantically aligned code from formal models. However, the ease of producing large volumes of code in a single prompt may tempt developers to skip manual checks, reinforcing the need for automated verification to ensure correctness and consistency.

Future work will explore novel domains to assess generalization and address integration and performance challenges. Key directions include advancing verification techniques, refining prompts, and improving context tracking for iterative development. Though unlikely to replace traditional MDD tools, LLMs can serve as powerful assistants when coupled with rigorous verification and validation frameworks.

REFERENCES

Bracha, G. (2016). *The Dart Programming Language*. Addison-Wesley.

Carnevali, L., D'Amico, D., Ridi, L., and Vicario, E. (2009). Automatic code generation from real-time systems specifications. In *IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 102–105, Paris, France.

Cheon, Y. (2021). Toward more effective use of assertions for mobile app development. In *IEEE International Conference on Progress in Informatics and Computing (PIC)*, pages 319–323, Shanghai, China.

Cheon, Y. and Barua, A. (2018). Model driven development for Android apps. In *International Conference on Software Engineering Research & Practice*, pages 17–22, Las Vegas, Nevada.

Cheon, Y., Lozano, R., and Senthil-Prabhu, R. (2023). A library-based approach for writing design assertions. In *IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 22–27, Orlando, FL.

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., abd S. Yoo, S. S., and Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems. In *IEEE/ACM International Conference on Software Engineering*, pages 31–53, Melbourne, Australia.

Flutter (2025). Build for any screen. https://flutter.dev/. Accessed: 2025-04-24.

France, R. B., Ghosh, S., Dinh-Trong, T., and Solberg, A. (2006). Model-driven development using UML 2.0: promises and pitfalls. *IEEE Computer*, 39(2):59–66.

Frantz, C. K. and Nowostawski, M. (2016). From institutions to code: Towards automated generation of smart contracts. In *IEEE 1st International Workshops on Foundations and Applications of Self-* Systems (FAS*W)*, pages 210–215, Augsburg, Germany.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., and Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79.

Jain, N., Vaidyanath, S., Iyer, A., Natarajan, N., Parthasarathy, S., Rajamani, S., and Sharma, R. (2022). Jigsaw: Large language models meet program synthesis. In *International Conference on Software Engineering*, pages 1219–1231, Pittsburgh, PA.

Jurgelaitis, M. et al. (2021). Smart contract code generation from platform specific model for Hyperledger Go. *Trends and Applications in Information Systems and Technologies*, 4(9):63–73.

Meservy, T. O. and Fenstermacher, K. D. (2005). Transforming software development: an MDA road map. *IEEE Computer*, 38(9):52–58.

Object Management Group (2014). Object Constraint Language, version 2.4. http://www.omg.org/spec/OCL/. Accessed: 2025-04-24.

PlantUML (2025). PlantUML at a Glance. https://plantuml.com. Accessed: 2025-04-24.

Sebastian, G., Gallud, J., and Tesoriero, R. (2020). Code generation using model driven architecture: A systematic mapping study. *Journal of Computer Languages*, 56:100935.

Stahl, T. and Volter, M. (2006). *Model-Driven Software Development*. Wiley.

Syriani, E., Luhunu, L., and Sahraoui, H. (2018). Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures*, 52:43–62.

Tsiounis, K. and Kontogiannis, K. (2023). Goal driven code generation for smart contract assemblies. In *International Conference on Software and Computer Applications*, pages 112–121, Kuantan, Malaysia.

Wang, J. et al. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936.

Warmer, J. and Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2nd edition.

Xue, J. et al. (2024). Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30.

Zheng, Z. et al. (2025). Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering*, 30(2):50.