Towards Agentic Human-in-the-Loop Model Transformation Rule Creation

MohammadHadi Dehghani[0009−0002−5540−5841], Stefan Klikovits[0000−0003−4212−7029], and Manuel Wimmer[0000−0002−1124−7098]

Johannes Kepler University Linz, Austria {mohammadhadi.dehghani,stefan.klikovits,manuel.wimmer}@jku.at

1 Introduction

Model Transformations (MTs) and Model-Based Optimisation (MBO) [6] serve as a core technology in Digital Twin (DT) engineering that enables the model-driven evolution and improvement of system configurations. MOMoT [3], for instance, is a tool that enables the search-based identification of MT sequences to optimise certain objectives.

Problem Description A common difficulty of applying MBO, however, is the complex creation and correct configuration of the set of MT *rules* that describe the way a (valid) model may be transformed without sacrificing validity. Past approaches to facilitate MT rule specification aimed to automatically learn them using examples [2,5], and use visual graph representations to simplify rule definition (e.g., Henshin [1]).

Nonetheless, the creation of a set of well-configured MT rules typically remains a complex trial-and-error process that demands experience and expert knowledge. This hampers the application of MT and MBO and demands the collaboration of domain experts with MT engineers.

To lower the entry bar, our research proposes a Human-in-the-Loop (HiL) setup for the Artificial Intelligence (AI)-supported generation of MT rules. More specifically, we connect an LLM-based authoring agent to a three-tiered Henshin validator (structure, semantic, applicability) in a human-supervised feedback loop. The user supplies a metamodel, example model, and natural-language description of change intent to enable an AI-agent to draft Henshin rules. Failed checks trigger guided revision, and approved rules enter the reusable library.

By combining the automated exploration of Agentic AI with a user-driven HiL setup, we enable an iterative workflow that facilitates the wider adoption of MBO among non-expert end users.

2 LLM-based MT Rule Generation

Our initial experiments with LLM-based one-shot rule generation using Henshin [1] showed that the current generation of LLMs is not able to produce satisfactory MT sets of rules directly.

In particular, we observed that generated Henshin rules via LLM usually show failure in at least one of the following three levels.

- On the **linguistic level**, generated Henshin modules are sometimes not well-formed `XMI` files. Some modules are missing declarations, may have invalid types or broken internal references, and sometimes they have invalid Left Hand Side (LHS) or Right Hand Side (RHS) definitions.
- On the **ontological level**, even for correct `XMI` files, oftentimes type references do not resolve the target EClass, EAttribute, or EReference of the metamodel. Moreover, we observed non-resolvable `nsURI` imports, and node types pointing to hallucinated targets.
- At **execution time**, a portion of the rules that passed on the former two levels failed to execute as intended on the given input model instances, producing no change due to too restrictive or non-applicable LHS, or resulting in output models that violated metamodel constraints.

In short, in our experiments, a pure LLM approach could only generate valid Henshin rules in very simplistic scenarios, but it could not reliably generate rules that are syntactically valid, semantically correct, and result in the intended output models.

To overcome this difficulty, we envision enhancing the LLM-only approach through the creation of an agentic AI setup, where LLMs have access to a range of tools, including model execution. This combines the best from both symbolic AI and non-symbolic AI, where agentic LLMs are used for rule generation, while validation is executed through MDE operations, which provide precision and robustness.

3 Methodology

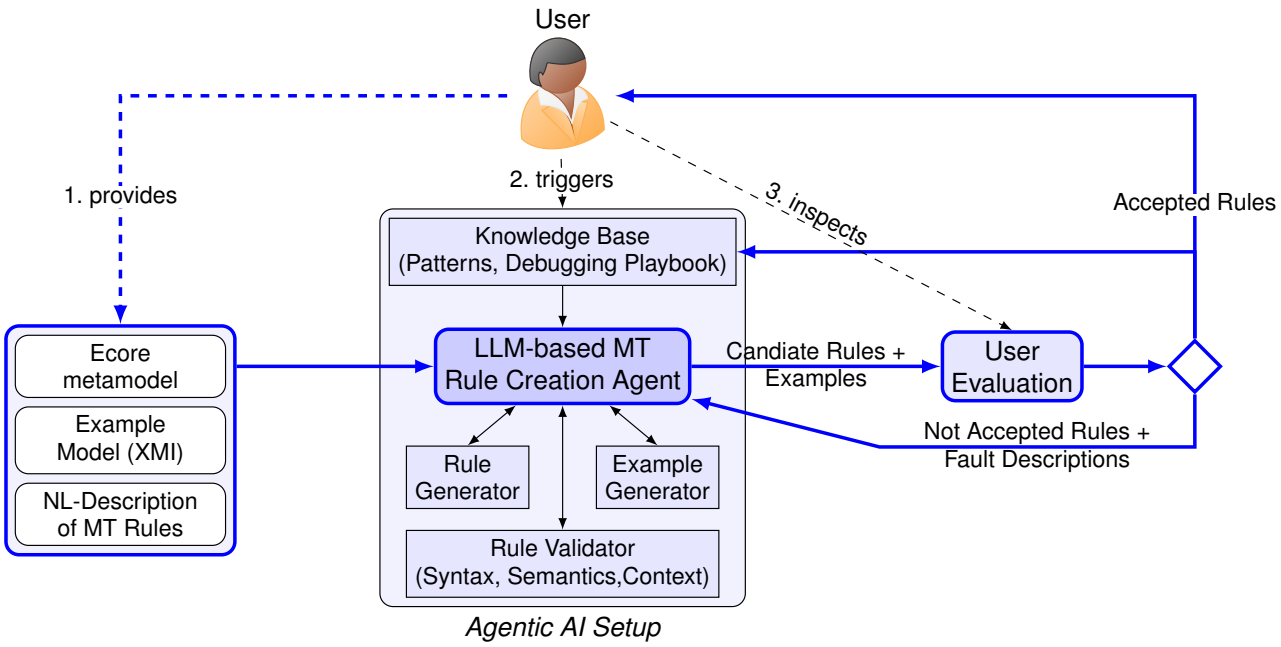


Fig. 1. Schematic representation of the agentic MT rule creation workflow. The user provides the inputs to the LLM-based MT rule creation agent, which has access to a knowledge base and interacts with various tools. Generated candidate rules and matching transformation examples are presented to the user for evaluation. Accepted rules are output, while not accepted ones are fed back to the agent alongside fault descriptions.

Our envisioned methodology treats MT rule construction as an iterative, human-supervised process rather than one-shot code generation (Figure 1). The user 1. provides a metamodel, an XMI example model, and a natural-language description of the intended change that the MT rule is supposed to perform. This 2. triggers an LLM-based agent to analyse the metamodel and the example model and proposes validated candidate MT rules to the user. The user remains active in the loop, 3. inspecting the generated rules, refining their intent, and accepting/rejecting the candidate rules.

Within the generation step, the agent generates candidate rules and only proposes each one to the user after it has successfully passed the three-tiered checks in the `Rule Validator`. These checks include structural well-formedness, semantic correctness based on the target metamodel, and validating the output of the rule when applied to an example input, ensuring the applicability and validity of the resulting model.

Some issues, such as invalid or too restrictive matching patterns, inapplicability on real model instances, or unintended output models are usually detected too late at execution time. The reason is that a test execution is required for these issues to surface. That is why in our approach, test input models are tailored by the LLM to bring fault detection forward to the authoring phase. A Henshin MT patterns wiki and debugging playbook that covers the most common patterns and mistakes guide the agent to produce correct, high-quality MT rules. The agent consults the user to make sure their intention is completely captured and reflected in the generated rule.

4 Prototype Implementation

We implemented a standalone agentic workflow capable of generating Henshin MT rules[1] to test our proposed methodology. It combines high-level agentic planning written in Node.JS and low-level EMF/Henshin validation written in Java. It consists of three main components: the *Three-Tier Validation Engine*, *Agent-Optimized Wiki Knowledge Base*, and an *Iterative Self-Correction Loop*.

[1] https://github.com/hadiDHD/henshin-agent

Three-Tier Validation Engine A command-line, Java-based validation engine to find fundamental syntax errors that follows a strict, fast-failing sequence of validation tiers:

1. **Tier 1: Structural Validation:** A lightweight check that has the sole purpose of verifying if the generated rule is a well-formed XMI document and conforms to the Henshin metamodel schema. That means it will make sure that all nodes, edges and units have distinct `xmi:id` attributes and proper boundaries in a valid XMI file.
2. **Tier 2: Semantic Validation:** Interoperates with EMF, registers the metamodel, loads the candidate Henshin rule module, and performs explicit checks to ensure that all type reference bindings (i.e., classes, attributes, and references) resolve to valid metamodel structural definitions.
3. **Tier 3: Applicability Validation:** Utilizes the Henshin standalone execution engine to try matching and applying the candidate rule on a real example model instance. If a match is found, the engine mutates the graph (ensuring that no EMF structural constraints are violated) and outputs the resulting model instance.

Agent-Optimized Wiki Knowledge Base We organize the project documentation in one `wiki/` directory, following Karpathy's *LLM Wiki Pattern* [4]. Having this structure maximizes information density and helps the LLM use less tokens. We include contrastive learning pairs (positive vs. negative examples), an explicit platform error-resolution matrix, and self-contained case-study references.

Execution Workflow This workflow is driven by an LLM, which invokes file-system tools to read both the metamodel and example model, queries the wiki for relevant transformation patterns, and returns candidate Henshin files as output. Upon receiving this condition, the agent executes a validator script. In case of an error reported by the validator, the agent intercepts the validator message, queries the wiki playbook for resolution to this specific failure, revises the candidate rule and re-validates. Only rules that succeed in all three levels are recommended to the user.

Human-in-the-Loop Interaction The user has two points of interaction with the system along the workflow. The user starts by calling the process providing 3 artefacts: an Ecore metamodel, an example XMI instance, and a natural-language description of the intended transformation. After that it is on auto-pilot by virtue of a self-correction and validation loop. Secondly, when a candidate rule passes the three levels of validation, it is passed to the user alongside the particular input example model and output model instance. A human looks at the output of a proposed rule, accepts it (for placement in a reusable library) or rejects it (providing an explanation of why this is so), and the fault description gets turned into feedback to feed to the agent for another round of trying to create a correct rule.

5 Next Steps

Following the creation of our proof-of-concept prototype, we will extend our work in several directions. First, we will setup a thorough evaluation setup to analyse the performance of our tool on known benchmark problems, as well as purpose-built, new problems to avoid training data bias. This allows an ablation study, showing our approach's advantage empirically. Second, we aim to extend our workflow to integrate more capabilities. For instance, we plan to add an LLM-as-a-judge component into the workflow, such that the user is less frequently asked. Finally, we aim to increase the complexity of the generation step and, rather than individual rules or small rule sets, create entire MT suites that cover all important aspects of a model-metamodel pair. This will require the extension of our pattern wiki by entire MT pattern suites taken from realistic example systems.

Acknowledgments. This research was funded in whole or in part by the Austrian Science Fund (FWF) projects 10.55776/PIN3686425 and 10.55776/PIN1684325.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place emf model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) Model Driven Engineering Languages and Systems. pp. 121–135. Springer (2010)

2. Brosch, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: An example is worth a thousand words: Composite operation modeling by-example. In: Schürr, A., Selic, B. (eds.) Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5795, pp. 271–285. Springer (2009). https://doi.org/10.1007/978-3-642-04425-0_20, https://doi.org/10.1007/978-3-642-04425-0_20
3. Fleck, M., Troya, J., Wimmer, M.: Search-based model transformations with momot. In: International Conference on Theory and Practice of Model Transformations. pp. 79–87. Springer (2016)
4. Karpathy, A.: Llm wiki. https://gist.github.com/karpathy/442a6bf555914893e9891c11519de94f (April 2026), gitHub Gist. Last Accessed: 19 Jun. 2026
5. Langer, P., Wimmer, M., Kappel, G.: Model-to-model transformations by demonstration. In: International Conference on Theory and Practice of Model Transformations. pp. 153–167. Springer (2010)
6. Zschaler, S., Mandow, L.: Towards model-based optimisation: Using domain knowledge explicitly. In: Software Technologies: Applications and Foundations (STAF 2016). Lecture Notes in Computer Science, vol. 9946, pp. 317–329. Springer (2016). https://doi.org/10.1007/978-3-319-50230-4_24