Early, Non-Blocking Validation for Agile Model-Driven Engineering Using Python*

Shirin Hussein[1,*], Sobhan Y. Tehrani[1]

[1]University College London (UCL), Gower St, London WC1E 6BT

Abstract

Model-Driven Engineering (MDE) usually assumes that models must be complete and correct before they can be analyzed or turned into code. This works well when a system is relatively stable, but it does not fit very well with agile development, where teams iterate quickly and expect feedback early. In practice, engineers often work with models that are only partially finished or are constantly changing, and most existing MDE validation tools do not handle that very well. For example, model validation frameworks that use Object Constraint Language (OCL) or Eclipse Modelling Framework (EMF) usually require models to be complete and well-formed before checking constraints. If there are violations, they are often treated as errors that stop further processing. This approach is less suitable for models that are built in steps or are only partly specified. We introduce a fail-fast validation approach that enables early, non-blocking analysis of incomplete models, addressing a key limitation of existing MDE validation techniques. We propose a novel fail-fast validation approach for Agile Model-Driven Engineering (Agile MDE) that leverages lightweight Python-based rules as an alternative to traditional OCL/EMF-based validation. Instead of blocking progress until a model is "correct," our approach runs simple checks early on to catch likely problems and give useful feedback as the model grows. These checks are lightweight Python functions, which makes them easy to write, understand, and update as both the models and requirements change without needing heavy MDE tooling. We show how this works using two realistic case studies in Digital Signal Processing (DSP) modelling and embedded software configuration. The examples illustrate how early validation can identify design issues sooner and support faster iterative refinement compared to traditional validation approaches. We demonstrate that fail-fast validation improves flexibility and developer feedback in Agile MDE while complementing, rather than replacing, traditional validation techniques.

Keywords

Model-Driven Engineering, Agile Development, Model Validation, Fail-Fast Validation, Python

1. Introduction

Model-Driven Engineering (MDE) aims to enhance software development by utilizing models to increase the level of abstraction and automate tasks. MDE methods usually expect models to be complete and consistent before analysis or code generation [1, 2]. While ensuring correctness is important, this approach often conflicts with agile practices, which emphasize rapid cycles, early feedback, and incremental development [3, 4]. In practice, users often encounter incomplete or modifiable models, particularly in the early stages. Details like numeric ranges, timing, or resource limits might not be set yet, but teams still need to make design choices. Standard MDE validation tools usually give blocking errors for incomplete models, which delays feedback and makes agile work harder. Agile MDE seeks to reconcile these tensions by adapting MDE techniques to better support iterative development. Existing approaches either require complete models or rely on incremental validation techniques that still assume well-formedness. Existing approaches provide limited support for validating incomplete models without blocking progress or providing useful feedback early in development [5]. This paper introduces a fail-fast validation framework for Agile MDE that enables systematic, non-blocking validation of incomplete models using lightweight Python-based rules. The framework provides early, non-blocking feedback through actionable warnings that tolerate missing information. In this paper, "fail-fast" refers

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29-July 3, 2026

*Corresponding author.

†These authors contributed equally.

shirin.s.hussein@ucl.ac.uk (S. Hussein); sobhan.tehrani@ucl.ac.uk (S. Y. Tehrani)

0000-0003-0521-4801 (S. Hussein); 0000-0003-4417-0477 (S. Y. Tehrani)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-26

CEUR-WS.org/Vol-4338/AgileMDE-short1.pdf

to quickly identifying issues and providing immediate feedback, rather than terminating execution. Validation does not block progress and allows for ongoing improvement. Using Python instead of specialized constraint languages like OCL allows for flexible validation logic that fits easily into current development workflows and can adapt as needs change [6, 7]. The paper shows how this method works with two case studies: one on a fixed-point Digital Signal Processing (DSP) pipeline and another on configuring embedded software. These examples show that early, non-blocking validation can reveal numeric and resource risks sooner in development, helping teams work through faster cycles in different fields. The contributions of this paper are the following:

- A fail-fast validation framework for Agile MDE through early, non-blocking validation of partial models.
- A Python-based validation model for defining and evolving validation rules without heavyweight MDE tooling.
- An evaluation strategy based on two case studies (DSP modelling and embedded configuration) demonstrating earlier issue detection and improved iterative development.

2. Background & Motivation

MDE usually requires models to be correct and complete before they can be analyzed or used to generate code [8]. Many MDE tools use strict validation rules and show errors if something is missing or inconsistent. This helps keep models high-quality, but it also assumes they are already stable and clearly defined. In agile development, models change quickly as teams work in short cycles [9]. Early-stage models are often incomplete, yet developers still require feedback to identify design risks. Traditional validation treats incomplete models as invalid and produces blocking errors that interrupt the workflow [10]. This can slow iterative development and discourage modelling in early design stages [11]. To address this, this work explores a fail-fast validation approach that accepts partial models and provides early feedback. Instead of blocking errors, the system reports warnings so that developers can refine models gradually while still receiving automated guidance.

3. Fail-Fast Validation Approach

This section explains the proposed fail-fast validation method for Agile MDE. It aims to give early feedback on models as they develop, without requiring premature completeness or correctness.

3.1. Design Principles

The approach is guided by four main principles:

- **Partial models are first-class citizens:** A partial model is defined as a model $M = (E, A)$ in which some attributes are assigned uncertain values. Such models are considered valid, but validation outcomes may carry reduced confidence [12, 13].
- **Early, non-blocking validation:** A non-blocking validation process is defined as a function $V(M) \rightarrow R$, where R is a set of validation results that do not prevent further model evolution regardless of detected issues. This way, model development is not blocked in early stages [14].
- **Lightweight implementation:** The validation rules are lightweight Python functions that work with basic data structures. There is no need for complex modelling frameworks or metamodels. This matches recent works that support simpler and more flexible methods for MDE and tooling [8, 13].
- **Actionable feedback:** Each validation result is accompanied by a clear message and a suggestion to guide the next steps [3, 5].

3.2. Model Representation and Validation Rules

Models are represented using lightweight JSON structures (e.g., dictionaries in Python). These models are intentionally typeless, meaning they do not conform to a predefined metamodel or schema. This design choice enables flexibility when working with partial or evolving models, as elements and attributes can be added incrementally without requiring upfront structural completeness. This aligns with prior work advocating lightweight and flexible modelling approaches [9]. As a result, validation rules are responsible for interpreting structure and handling missing or optional fields. This approach does not use a predefined metamodel, which helps reduce setup costs and enables faster changes. However, there are some trade-offs. Unlike metamodel-based methods, this framework offers weaker structural guarantees, less automatic type checking, and less support for cross-language validation. Because the rules operate directly on JSON structures, the system's expressiveness and flexibility depend on the quality of the Python rules. This makes the approach useful for early agile validation, but it is not meant to replace metamodel-based validation when you need formal consistency, language-wide constraints, or code generation guarantees. Since models can be incomplete or partly specified, validation rules only check the information that is available and allow missing fields [12]. Missing or undefined attributes are handled using uncertainty-aware rule execution. A rule first checks whether the attributes required for a specific condition are available. If a required value is missing, the rule does not fail. Instead, the rule might skip the dependent check, make a cautious partial estimate, or display an informational warning with reduced confidence. For example, if worst-case execution time (WCET) values are missing, CPU utilisation can be calculated as a lower limit. This approach means a rule only makes claims based on the attributes that are currently available. A validation rule is defined as a function $r : M \to R$, where each result $r_i \in R$ is a tuple $(id, severity, message, suggestion, confidence)$.

The *confidence* value reflects the completeness of the input model and the reliability of the inferred result. The rules are kept simple and local so they can run often as the model evolves. For example, they might spot possible numeric overflow in a DSP model or high CPU use in an embedded setup. Listing 1 shows a representative validation rule. The rule tolerates partial models by checking only available fields, emitting a warning rather than blocking execution, including lower-confidence warnings when model information is incomplete. When required inputs are missing, the rule assigns lower confidence to its output or produces uncertainty-aware warnings rather than skipping validation.

```python
def check_overflow(dsp):
    warnings = []

    if dsp.get("in_bits") is None or dsp.get("acc_bits") is None:
        return warnings

    gain_bits = dsp.get("gain_bits", 0) or 0
    required = dsp["in_bits"] + gain_bits + 1

    if dsp["acc_bits"] < required:
        warnings.append("WARN_OVERFLOW")

    return warnings
```

Listing 1: Example fail-fast validation rule for a DSP model.

The validation rules are written in Python, but this method is not limited to Python systems. The validation layer works separately from the main system and connects with models using standard data formats like JSON. This makes it possible to use the approach with systems built in other languages, such as C, C++, or Java, by using simple interfaces, scripting, or APIs. Python is chosen mainly because it is flexible and makes it easy to write validation logic, not because it is required for the system.

3.3. Fail-Fast Validation Workflow

As models develop, validation rules give early feedback without stopping progress [3]. The process is iterative: a user creates or updates a model, which might be partial [12], then runs validation rules. Warnings and messages help improve the model. To clarify how validation operates on partial models, Algorithm 1 outlines the fail-fast validation process.

Algorithm 1 Fail-fast validation of a partial model

Require: Partial model M, validation rules R
Ensure: Validation results V

$V \leftarrow \emptyset$
for all $r \in R$ **do**
    required ← attributes of r
    if all required attributes are available **then**
        result ← $r(M)$
        confidence ← high
    else if r supports partial evaluation **then**
        result ← $r(M)$ using available attributes
        confidence ← reduced
        add `INFO_PARTIAL` to V if relevant
    else
        result ← uncertainty warning
        confidence ← low
    end if
    add *result* to V
end for
return V

Figure 1 illustrates the fail-fast validation workflow used in this approach. Even when all fail-fast validation rules pass, model refinement may still continue because Agile MDE is iterative. Passing validation means that no currently detectable issue has been found in the available information, not that the model is final or complete. Within a sprint, users can update requirements, add missing attributes, change design alternatives, or prepare the model for later code generation. Therefore, refinement remains part of the loop to support continuous evolution in sprint iterations. This method works alongside traditional MDE validation. Quick fail-fast checks guide model development early on, while more thorough validation can be done once the model is more complete. The workflow can be integrated into continuous development pipelines, enabling automated validation on each model update and supporting rapid feedback cycles in agile environments.

4. Case Study

To demonstrate the proposed fail-fast validation approach, we apply it to two distinct domains: (i) a fixed-point DSP modelling scenario and (ii) an embedded software configuration scenario. These case studies aim to measure how well fail-fast validation helps detect issues earlier, shorten iteration cycles, and improve feedback compared to traditional validation methods.

4.1. Fixed-Point DSP Pipeline

The first case study considers a representative fixed-point DSP pipeline involving multiple stages, gain, accumulation, and optional saturation. Early in development, details such as bit-widths may not be set, but traditional MDE validation typically requires full specifications before analysis [1, 8]. With the proposed method, the pipeline is modeled partially and checked using lightweight Python validation

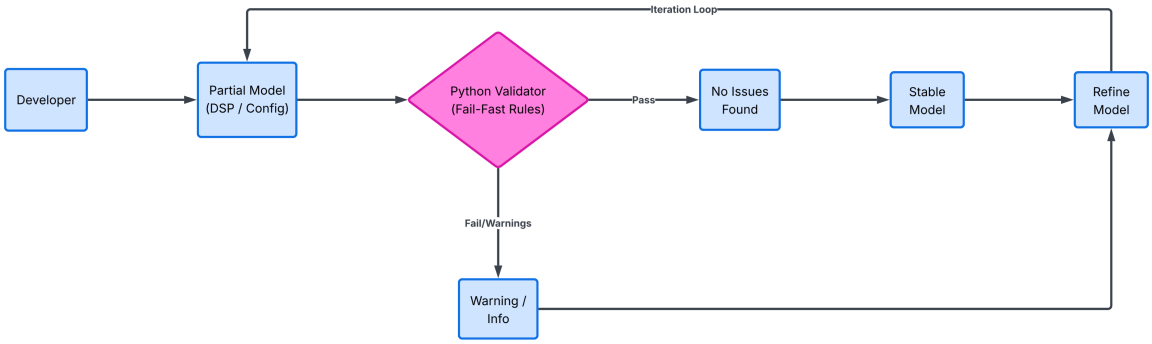


Figure 1: Fail-fast validation workflow. Partial models are analyzed by lightweight Python validation rules that produce non-blocking warnings, guiding iterative model refinement during agile development.

rules. In one early version, the model includes input and gain bit-widths and a tentative accumulator width, but omits saturation. The validation rules detect overflow risks and missing saturation in early iterations. These issues are identified before full model completion, reducing the number of refinement iterations required from three to one compared to a traditional validation workflow. In contrast, traditional validation would only detect these issues after full specification, delaying feedback and increasing rework. This shows that fail-fast validation provides early feedback as the model evolves, helping progress continue smoothly.

Listing 2: Initial DSP model used in the case study.

```
dsp_model = {
    "in_bits": 16,
    "gain_bits": 6,
    "acc_bits": 20,
    "saturation": False,
    "stages": 6
}
```

4.2. Embedded Software Configuration

The second case study examines an embedded software configuration model that describes periodic tasks and memory usage. Initial configurations may lack complete information, such as WCET values. Application of the validation rules generates warnings regarding high estimated CPU utilization, incomplete timing data, and potential memory constraints. These early warnings enable refinement of task parameters and resource allocation before full model completion, reducing late-stage validation errors compared to traditional approaches. Once missing parameters are completed and the configuration is adjusted, the warnings are reduced or eliminated. Alongside the DSP example, this demonstrates that the proposed fail-fast validation approach offers valuable feedback on partial models in various domains.

Listing 3: Initial embedded configuration model.

```
config_model = {
    "tasks": [
        {"name": "sensor", "period_ms": 10, "wcet_ms": 4},
        {"name": "control", "period_ms": 20, "wcet_ms": None},
        {"name": "logging", "period_ms": 50, "wcet_ms": 18}
    ],
    "memory": {"total_kb": 256, "used_kb": 245}
}
```

Listing 4: Example fail-fast validation rule for embedded configuration.

```
def check_cpu_utilization(cfg):
    warnings = []
    tasks = cfg.get("tasks", [])

    utilization = 0.0
    incomplete = False

    for t in tasks:
        wcet = t.get("wcet_ms")
        period = t.get("period_ms")

        if wcet is None or period is None:
            incomplete = True
            continue

        utilization += wcet / period

    if utilization > 0.7:
        warnings.append({
            "id": "WARN_CPU_UTIL",
            "message": f"Estimated CPU utilization is high (U={utilization:.2f}).",
            "suggestion": "Consider increasing task periods or reducing WCET."
        })

    if incomplete:
        warnings.append({
            "id": "INFO_PARTIAL",
            "message": "Some tasks have missing parameters; utilization is a lower bound.",
            "suggestion": "Provide missing WCET/period values."
        })

    return warnings
```

Listing 4 shows a validation rule from the embedded configuration case study. This rule estimates CPU usage using the available task parameters and can handle missing values by giving results with partial confidence. Rather than failing when some information is missing, it provides lower-bound estimates and helpful warnings, which allow for early feedback during model development. This rule corresponds to the WARN_CPU_UTIL and INFO_PARTIAL results reported in Table 1.

Applying the validation rules to the initial configuration produced warnings for high CPU utilization and high memory usage, together with an informational message indicating that utilization was only a lower bound because one WCET value was missing.

Table 1
Validation results for the two case studies with comparison to traditional validation.

Case study	Traditional validation (baseline)	Fail-fast initial result	After refinement
DSP pipeline	Validation deferred until model completion (no early feedback)	WARN_OVERFLOW, WARN_NO_SAT, WARN_LATENCY	No warnings
Embedded configuration	Incomplete parameters delay or prevent validation	WARN_CPU_UTIL, INFO_PARTIAL, WARN_MEM	Reduced or no warnings

Table 1 summarizes the validation results, highlighting early issue detection and their elimination after refinement across both case studies. The table also includes a comparison with traditional validation approaches, where validation is typically deferred until models are complete and well-formed. In both

case studies, the fail-fast approach detects critical issues earlier, before full model completion, whereas the baseline provides little or no feedback at early stages. This reduces the number of refinement iterations and helps avoid late-stage issues. Traditional OCL- or EMF-based validation usually requires models to be complete and well-formed. In contrast, our approach provides continuous, non-blocking feedback throughout model development.

5. Related Work

MDE has long focused on precise modelling, metamodelling, and strict validation to make sure models are correct before analysis or code generation [1]. Methods such as Model-Driven Architecture (MDA) [15] and metamodel-based validation frameworks help maintain model consistency and correctness by enforcing structural and semantic rules. These techniques operate well when development is stable, but they usually expect models to be complete and well-defined before validation. To better support iterative development, research in Agile MDE has looked at lighter modelling practices and faster feedback [9]. Earlier work has explored how to combine modelling with agile processes and reduce the complexity of traditional modelling tools. Some have suggested using general-purpose languages or scripting environments for lightweight modelling instead of heavier frameworks [12, 13]. Studies have also explored incremental and continuous validation to make validation more responsive as models change, focusing on quickly revalidating models after updates [5, 6, 7, 16]. As shown in Table 2, existing approaches either require complete models or provide limited incremental validation while still assuming well-formedness. In contrast, the proposed approach supports partial models, non-blocking validation, and a lightweight design. Unlike incremental validation approaches that still assume well-formed models, our method explicitly supports incomplete models and non-blocking feedback. This paper takes a different approach by allowing partial models and giving non-blocking feedback early in development. By writing validation rules as lightweight Python functions, our method aims for quick feedback and flexibility, working alongside traditional MDE validation techniques.

Table 2
Comparison of validation approaches in MDE

Approach	Handles Partial Models	Non-Blocking Validation	Lightweight Implementation
OCL-based validation	No	No	No
Incremental validation	Partial	No	Partial
Proposed approach	Yes	Yes	Yes

6. Conclusion

This paper examines a fail-fast validation approach for Agile MDE that delivers early, non-blocking feedback on evolving and incomplete models. Validation rules are expressed as lightweight Python functions, enabling users to identify potential issues early without the need for heavyweight modelling tools or fully specified models. Two case studies, a fixed-point digital signal processing pipeline and an embedded software configuration scenario, demonstrate that the approach reveals design risks such as numeric overflow, latency growth, and resource constraints during early development stages. The findings indicate that fail-fast validation can help improve iteration speed and flexibility in agile modelling workflows. While the evaluation is based on two representative case studies, they were selected to cover distinct domains (DSP modelling and embedded configuration), demonstrating the generality of the approach. Future work will extend the evaluation to larger-scale industrial models and quantitative comparisons with existing validation frameworks.

Declaration on Generative AI

The authors declare that no generative artificial intelligence tools were used to generate text, figures, code, data, analyses, or reviews for this submission.

References

[1] D. C. Schmidt, Model-driven engineering, Computer 39 (2006) 25–31.

[2] S. Kent, Model driven engineering, in: Integrated Formal Methods, volume 2335 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 286–298. doi:10.1007/3-540-47884-1_16.

[3] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al., Manifesto for agile software development, Snowbird (2001).

[4] K. Beck, C. Andres, Extreme Programming Explained: Embrace Change, 2nd ed., Addison-Wesley, 2004.

[5] J. Cabot, E. Teniente, Incremental evaluation of ocl constraints, in: Advanced Information Systems Engineering, Lecture Notes in Computer Science, Springer, 2006, pp. 81–95.

[6] J. Cabot, M. Gogolla, Object constraint language (ocl): A definitive guide, in: Formal Methods for the Design of Computer, Communication and Software Systems, Lecture Notes in Computer Science, Springer, 2012, pp. 58–90.

[7] M. Richters, M. Gogolla, On formalizing the uml object constraint language ocl, in: Conceptual Modeling, Lecture Notes in Computer Science, Springer, 1998, pp. 449–464.

[8] L. Burgueño, D. D. Ruscio, H. Sahraoui, M. Wimmer, Automation in model-driven engineering: A look back, and ahead, ACM Transactions on Software Engineering and Methodology 34 (2025) 1–25.

[9] K. Lano, H. Alfraihi, H. Haughton, Research directions for agile model-driven engineering, in: Software Technologies: Applications and Foundations, Lecture Notes in Computer Science, Springer, 2024, pp. 1–11.

[10] H. Alfraihi, K. Lano, Report from mde practice: An interview-based evaluation of model-driven engineering uses, PLoS ONE 20 (2025) e0335461.

[11] M. A. Umar, K. Lano, A. K. Abubakar, Automated requirements engineering framework for agile model-driven development, Frontiers in Computer Science 7 (2025) 1537100.

[12] M. Famelis, R. Salay, M. Chechik, Partial models: Towards modeling and reasoning with uncertainty, in: International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 573–583.

[13] M. B. Chaaben, L. Burgueño, I. David, H. Sahraoui, On the utility of domain modeling assistance with large language models, ACM Transactions on Software Engineering and Methodology (2024).

[14] K. Lano, H. Haughton, Z. Yuan, H. Alfraihi, Agile model-driven re-engineering, Innovations in Systems and Software Engineering 20 (2024) 559–584.

[15] Object Management Group, MDA Guide Version 1.0.1, Technical Report, Object Management Group, 2003.

[16] S. Hussein, S. Y. Tehrani, H. Beheshti, Model-driven engineering in digital signal processing, in: Proceedings of the STAF 2025 Workshops, Koblenz, Germany, 2025.