On the Integration of Formal Model Transformations with Manual Document Manipulations in Software Engineering Processes

Hans-Werner Sehring
Department of Computer Science
Nordakademie
Elmshorn, Germany
e-mail: sehring@nordakademie.de

***Abstract*—Model-driven software engineering processes are based on formal models that can be automatically transformed into one another after specifications are added. However, the creation of many software products involves creative activities that result in manually generated, informal documents, which prevent automated model transformations. To enable transformation steps, the content of these documents must be accessed in a structured way when integrating them into model-driven processes. Manually maintained documents are subject to frequent changes, including modifications to their structure. To enable model-driven processes in the presence of creative activities and their documents, we are currently experimenting with parsing techniques that combine the structure of documents with domain knowledge about their content. First experiments are based on the Minimalistic Metamodeling Language and its ability to integrate semantic descriptions with syntactic representations.**

Keywords-software development; software engineering; computer aided software engineering; top-down programming; document handling.

I. Introduction

Software engineering processes involve creating and consuming a series of documents. Such documents link the various phases of activity in software creation processes, whether experts perform sequential work in phase-oriented projects or cross-functional teams collaborate simultaneously using agile approaches.

In general, documents may contain various types of content, such as problem statements, requirements, constraints, domain models, solution models, abstract and concrete descriptions of (software) implementations, tests cases, data, configurations, design decisions, specifications of development and quality assurance processes, and user and maintenance manuals.

The documents' formats are diverse. They include text documents, figures, (states of) collaborative digital whiteboards, interactive presentations, prototypes, and workshops protocols (including photos of whiteboards, for instance).

As such, managing the series of documents created during a (software) development process resembles content management tasks.

The way documents are handled depends on the software development approach taken.

- Documents may be manually created, human-perceivable representations of content. This is how documents are managed in human-centered processes, particularly creative ones.
- Documents may also represent formal models. In this case, formality allows for automated transformation steps in software development and for explicit traceability. This notion of documents is found in model-driven approaches. Documents may also be generated from models as they are in content management processes.
- Dialogs between a user and a *Generative Artifical Intelligence* (*GenAI*) system can also be considered documents. Each dialogue consists of a series of prompts used as input for the GenAI, as well as the AI's responses to these prompts. The utilization of GenAI in software engineering processes is currently being researched, for instance, under the term *vibe modeling*.

Combinations of these software development methods may be of interest. This article presents a step toward integrating of creative manual work on documents into model-driven processes. It provides an extension of the work presented in [1].

The class of software engineering processes that are based on documents that contain formal models are called *Model-Driven Software Engineering* (*MDSE*) or *Model-Driven Software Development* (*MDSD*) processes [2].

Software engineering processes that include creative activities, such as conceptual modeling or interaction design [3], are often applied for interactive software. Creative activities are supported by documents that have neither a common format [4] nor formal semantics. Instead, they reflect subjective impressions, case-based presentations, alternatives, and similar content directed at a human audience.

Documents that lack formal structure cannot participate in MDSE processes per se. However, they can be annotated by their creators with, for example, with references to relevant content that are sufficiently fine-grained to address well-formed content. Such annotations allow creative documents to participate in MDSE processes.

However, such annotations refer to specific document *instances*. Documents used in creative activities are, in particular, working documents that are subject to constant change. This includes changes in the structure of the documents. Therefore, any fixed reference to content in such a document will potentially become invalid and metadata may become inconsistent as work progresses.

In this article, we investigate means of integrating informal documents, in particular ones that are subject to change,

into (model-driven) software engineering processes. We are currently experimenting with linguistic means of recognizing the content of documents with changing structures. First experiments with document recognition are based on a modeling language and its special ability to integrate semantic descriptions with syntactic representations.

Preliminary results show that at least some content can be extracted from documents that lack formal representations. In this way, model-driven approaches can potentially be applied to software projects with creative aspects.

GenAI allows another take on the problem. It is specifically suited to generate formal representations from natural language descriptions. MDSE approaches based on GenAI are beyond the scope of this article.

The remainder of this article is organized as follows: In Section II, we revisit model-driven software engineering and discuss the need for incorporating informal documents. Section III presents typical ways of referencing content in single documents, and it addresses means of managing volatile references to content of mutable documents. Section IV briefly introduces a modeling language that is used for initial experiments in this article. An experimental implementation of these concepts is presented in Section V. The article concludes in Section VI with a summary and an outlook on future work.

II. Visual Software Engineering Artifacts

The discourse in this article does not require a comprehensive introduction to model-driven approaches. However, this section introduces some basic terms and highlights the challenges of integrating creative work.

A. Model-Driven Software Engineering

In software development processes, a series of documents is created. The kinds of documents may differ depending on the kind of software being created and on the methodology used for the process. But all documents serve common purposes, such as linking activities by the results represented in them, allowing traceability of activities [5], and others.

MDSE formalizes the flow of documents and thus the connection of development steps. Documents are *models* with a formal semantics. Models are derived by means of *model-to-model transformations* and finally to code in *model-to-text transformations* on a (semi-) automatic basis. This way, development steps can be performed (semi-) automatically and changes to models can be propagated down the model chain.

One of the first prominent examples of MDSE is the Object Management Group's *Model-Driven Architecture* (*MDA*). Various other approaches have emerged that differ in the way in which they implement transformations, for example, by means of metaprogramming [7], code templates [8], or GenAI [9].

B. Creative Software Development Activities

Certain kinds of software solutions, such as those with a focus on the human-machine interface, include creative steps. Examples of creative activities are the definition of interaction patterns, of user experience in general, and user interface design in particular [10].

In [11], we use the term *Model-Supported Software Creation* (*MSSC*) to distinguish this kind of software development from general MDSE that relies purely on formal representations.

Figure 1 shows typical groups of artifacts created to model aspects of a software solution. Each group of artifacts belongs to a *modeling stage*, depending on the chosen development approach. Note that the sequence of artifacts is not meant to prescribe a temporal order. Depending on the development process, the artifact groups relate to consecutive activities, or they are just a classification of the outcomes of activities that are performed in an interleaved and, eventually, iteratively manner.

A first set of artifacts, in Figure 1 called *Preparation*, reflects the decisions to be made before starting a software development project. This includes identifying the problem, deciding to start a project to solve it, concluding that software is part of the overall solution, setting concrete project goals, and allocating resources.

The artifacts in the *Concept* stage take the perspective of the application domain. A *Domain model* will finally capture that application domain. Creative projects often start with *Personas*, stereotypical users for whom the software will be designed. *Customer Journeys* A *Solution hypothesis* gives a first idea of the software to be developed as part of an overall problem solution.

Typical creative processes, such as User-Centered Design or Design Sprint, rely on all stakeholders taking part in the design process. Since a discussion on the basis of models is too abstract for many domain experts, a series of prototypes that visualize the ideas is employed. Prototypes of different development stages are often called *Lo-Fi Prototypes* and *Hi-Fi Prototypes*. Low fidelity prototypes emphasize the information structure (*Information architecture*) and the rough layout of the software's use interface (*Wireframe*, *Module catalog*, *Navigation*). High fidelity prototypes provide a preview of the software to be developed based on the current insights. For this, the look of the software's user interface is designed in detail (*Style guide*) and some functionality is provided in usable or emulated form (*Click dummy*). The goal is to validate the solution hypothesis in user experiences tests (*UX Tests*).

Solution Architecture provides the transition from the domain perspective to a technical perspective. In includes a first idea of the software's structure, its interface, the required infrastructure and other first implementation decisions. The artifacts created to define the solution architecture depend on the kind of software to be developed. In ecommerce applications, for example, the customer journeys and the touchpoints along each journey provide valuable input. From these, the demand for functionality and data is derived by a *Touchpoint-data mapping* and a *touchpoint-function mapping*. These lead to a *High-level architecture* of main components and the required processes and data flows.

To complete the switch to a technical perspective, the design activities for a *Software Architecture* and the implementation

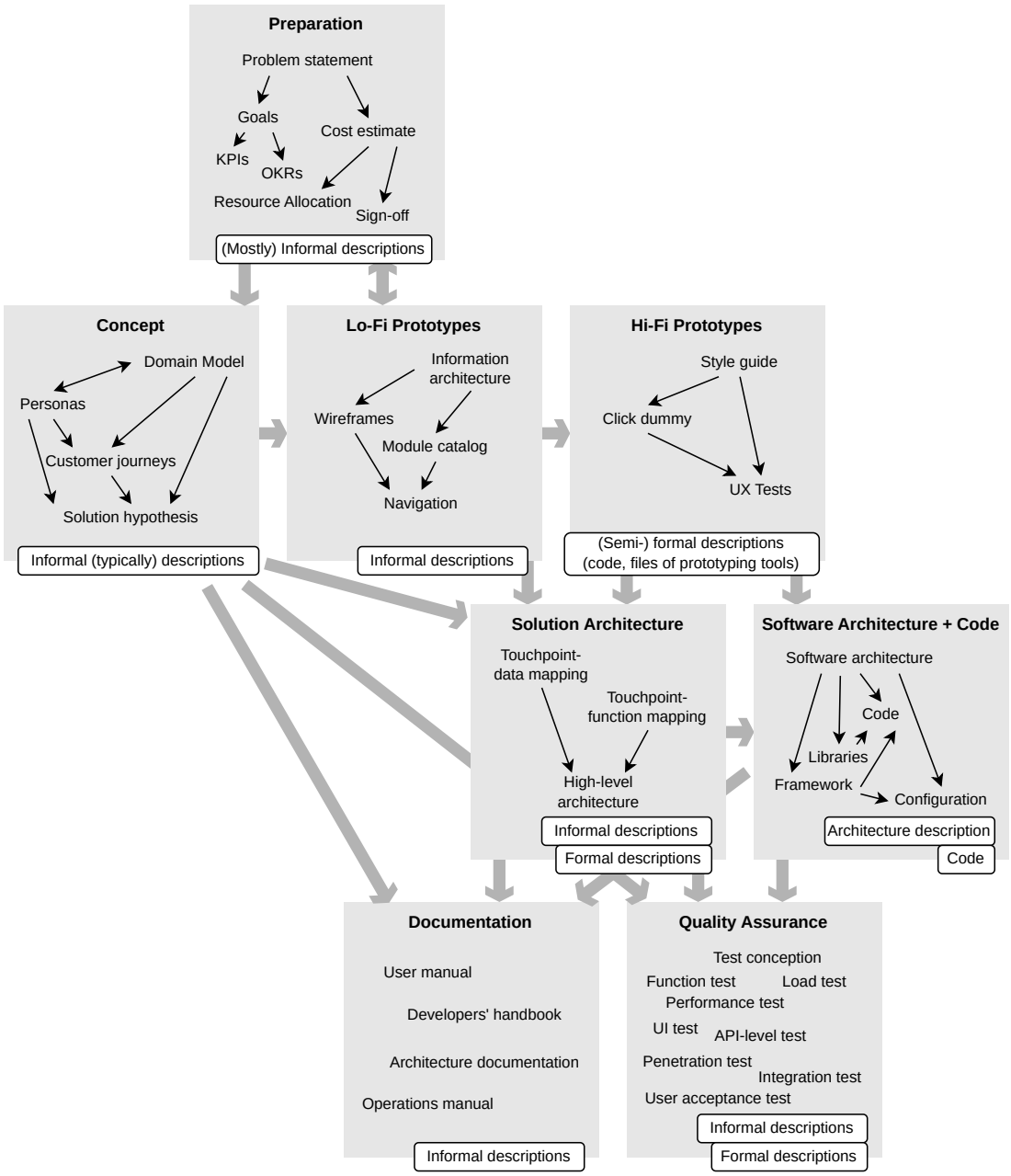


Figure 1. A typical flow of artifacts in a software development process that integrates creative activities, based on [6].

activities leading to *Code* elaborate the models to the point of operable software. There are different definitions of *Software architecture*. In particular, it explains how requirements are adressed in detail. When implementing the software design, *Code* is produced, incorporating *Libraries* and *Framwork*s. Software (also) consists of *Configuration*s to a varying degree. The degree depends on the implementation approach chosen: custom development or off-the-shelves software; software that is manually coded, generated, or built in a low code environment.

On top of the software itself, accompanying artifacts such as the various *Documentation* items and *Tests*, more specifically test concepts, test cases, and test scripts, are created as part of a software engineering process.

Models such as the *Domain model*, the *High-level architecture*, and the *Software architecture* can typically be expressed in a suitably formal way as to be derived from each other by model-to-model transformations. However, other documents are typical representatives of informal documents, such as *Personas*, *Customer journeys*, and *Style guides*. There may even be dynamic artifacts, such as a *Click dummy* that needs to be experienced by a human observer who interacts with it.

Informal documents are authored using tools with a focus on graphical presentations. Typical tools are presentation software, collaborative digital whiteboards, issue trackers, and the usual text processors and drawing software. During creative processes,

even structured entities, such as user stories and task boards are often represented in unstructured documents like whiteboard drawings.

C. Creative Artifacts in Model-Driven Processes

Depending on the type of software, there are different steps in the development process that are of an informal nature. Some software solutions require creative development activities. Typical such activities are those from the disciplines of domain modeling, conceptual modeling, and visual design. Such development steps are typically performed manually and lead to subjective results. As a result, tools that support creative activities often produce informal representations and documents. Therefore, software projects that involve creative activities cannot be fully covered by model-driven processes in most approaches.

In order to include creative activities in model-driven processes, the informal documents that are generated have to be interpreted in such a way that their content can be referenced and can be extracted in a defined structure. Through such an interpretation, content may be used in software models or during model transformations.

Interpretations of documents that lack formal structure can be added explicitly. For example, their creators may provide annotations with *content references* and *metadata* to guide access to relevant content. Such annotations, however, refer to specific document instances.

Creative activities typically consist of numerous iterations. As a consequence, documents used in creative activities are subject to constant change. This includes changes to the structure of the documents themselves. Therefore, any fixed reference to content in such documents will potentially become invalid and metadata may become inconsistent as work progresses. As a consequence, documents are required to be constantly reinterpreted.

Due to possible structural changes, the (re-) interpretation of documents must be defined based on some rules or patterns, such as grammars with syntactic and semantic rules.

III. Referencing Content in Documents

In order to extract content from documents in a form that is suitable for use in a formal development process, parts of that content must be addressable. This requires documents to be structured, or to allow superimposed structures for content references.

Digital documents can be structured to varying degrees. Typically, document formats are categorized as *structured*, *semi-structured*, and *unstructured*.

A. Structured Documents

Structured documents are created according to a well-defined structure, allowing them to be precisely analyzed. This can be realized in three different ways. First, the structure of documents may be used to query for content, such as object paths based on JSON definitions. Second, specific parts of a document can be addressed if structure elements have stable names (paths) or stable IDs. A third approach uses grammars which can be used to both create documents of a certain form and to parse documents to identify structural elements according to linguistic constructs.

A common structure to which multiple documents conform calls for a schema or document format. Schemas of structured documents differ in the meaning they convey. A format may reflect visual layout like, for example, in the case of HTML, it may use a generic semantics like, for example, XML formats for formal languages, or it may carry domain knowledge as, for example, application-specific XML formats.

B. Semistructured Documents

Documents that have a recognizable structure, but no common schema to which they conform, are called *semistructured*. Any interpretation rules applied to such documents are fragile in the sense that they may not be applicable to all document instances, or else all possible forms of documents must be considered in all rules.

If there is some technical structure that allows referencing parts of a document, then some pragmatics can be applied to interpret combinations of structure elements and content. For example, in a text document, there may be a recognizable structure of single-line terms written entirely in bold font. That structure may be interpreted as the term being a section heading. If the document is a software architecture description, and if the term is interpreted as a subsection of a section “Software Components”, then the term may be interpreted as the name of a software component.

In this way, semistructured documents are required to expose some recognizable structure. As these examples demonstrate, interpreting them requires some defined domain semantics and pragmatics in order to apply interpretation rules. This includes both domain entities, processes, constraints, etc., as well as typical representations and document layouts that are used in the domain. Note that the term “domain” refers to different levels of abstraction, ranging from a topic domain to a specific project.

C. Unstructured Documents

Unstructured documents, exhibit no structure that would allow referencing parts of a document. Typical examples of such documents are media files in binary format.

To reference parts of an unstructured document, some technical ways of addressing can be used, for example, pixel ranges in an image or timecode sequences in movies. Such references depend on the concrete document or, more precisely, on the actual presentation of it. For example, areas of an image that are defined by pixel coordinates relate to the resolution of that image. Such references are, therefore, volatile. For example, a selection of pixel coordinates is not valid for an equivalent image in different resolution.

There is no precise way to semantically reference content, although the semantics of unstructured documents can be analyzed by various algorithms.

D. Aggregating Documents from Different Sources

When accessing document collections that originate from different sources, the problem of different or varying schemas may arise. A typical approach to cope with such a situation is employing adapter components that allow accessing structured documents according to a common schema or by transforming them into a common schema [12].

However, when multiple documents describe the same domain entity, a common schema is not sufficient for precise retrieval of that content [13].

E. Extracting Content from Mutable Documents

As mentioned earlier, documents created during creative activities in software engineering processes are subject to change, which means they have to be *mutable* (*volatile*, sometimes called *living* documents).

In MDSE processes, the contents of documents are used to create software models from them, or such models are in other ways related to the contents of documents. Changing documents can generally break such relationships.

One solution is to create copies of documents once they are referenced and to keep these copies stable. But this would exclude further work on those documents from the process.

Parsing is a standard approach to identifying meaningful content in a document. For formal languages, a parsing process operates on the syntactic structures of a document and applies a defined semantics to interpret those structures.

Interpretation is driven by *parsing rules* that are part of a *grammar*. First, these rules are applied to recognize syntactic structures. Once syntactic structures according to a grammar have been detected in a document, a subsequent semantic analysis assigns meaning to those structures. Compilers for programming languages create an *Abstract Syntax Tree* (*AST*) during syntactic analysis and attribute its nodes during semantic analysis to derive a *decorated* AST. Similar techniques can be used for documents.

Documents resulting from creative processes do not follow a fixed semantics. Therefore, classical parsing approaches based on formal languages alone do not work on them. In our current research, we are using domain knowledge to augment document parsing.

Parsing of semistructured documents requires pragmatics since not all parts of a document have an identifiable structure. An open question is whether pragmatics can be provided by domain knowledge: two equally formatted expressions may be distinguished by some significant content. In general, domain knowledge may be necessary to decide on a parsing strategy.

Parsing is well understood for formal and, to a limited extent, semistructured representations, but it is usually applied once. Updating models based on subsequent parsing results of a modified document requires, according to our current findings, an additional relationship between document structure and domain semantics.

Updates can change a document on two levels: If a document's content is changed, then the document needs to be reparsed and reinterpreted. Detected changes lead to updates of subsequent models (documents).

If changes affect the structure of a document, then additional actions are required. When the interpretation of structures changes, the underlying parsing rules generally need to be updated because it is impossible to anticipate all possible structures in a grammar.

Apart from classical parsing techniques, GenAI performs document analysis.

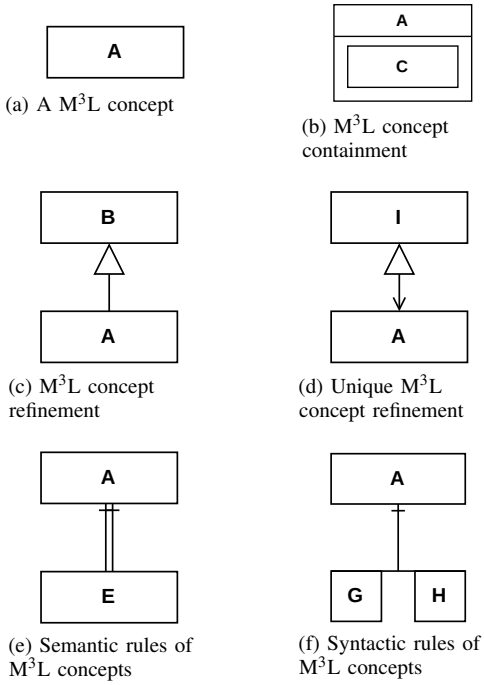


Figure 2. A graphical notation of M^{3}L concepts.

IV. The M^{3}L as a Modeling Language

The *Minimalistic Metamodeling Language* (*M^{3}L*) is a metamodeling language. As such, it can be applied to various modeling tasks. We use the M^{3}L for initial experiments in document recognition, capturing both domain semantics as well as document formatting.

This section provides a short introduction to the M^{3}L to introduce the fundamentals of the modeling approach and to illustrate its application to MDSE through some modeling patterns.

A. A Short Introduction Into the M^{3}L

The M^{3}L allows defining and deriving *concepts*. Definitions are of the general form

```
A is a B { C is a D } |= E { F } |- G H .
```

Such a statement matches or creates a concept *A*. All parts of such a statement except the concept name are optional.

In the course of this article, we use a graphical notation of the M^{3}L as shown in Figure 2 for the different parts of a concept definition. For concept refinement we borrow notation

```
ConditionalStatement is a Statement {
  Condition     is a Boolean
  ThenStatement is a Statement
  ElseStatement is a Statement
}
```

Figure 3. Sample base model of procedural programming.

```
IfTrueStmt is a ConditionalStatement {
  True is the Condition
} ⊨ ThenStatement

IfFalseStmt is a ConditionalStatement {
  False is the Condition
} ⊨ ElseStatement
```

Figure 4. Sample semantics of conditional statements.

from the *Unified Modeling Language* (*UML*), see Figure 2c for *is a* relationships and Figure 2d for *is the* relationships.

The concept *A* is a *refinement* of the concept *B*. Using the "is the" clause instead defines a concept as the only specialization of its base concept. Conflicting "is the" specializations are considered an error.

The concept *C* is defined in the *context* of *A*; *C* is part of the *content* of *A*. Contexts define (hierarchical) scopes. Concepts, such as *A* are defined in an unnamed top-level context.

There can be multiple statements about a concept visible in a scope. Statements about a concept are cumulated. This allows concepts to be defined differently in different contexts.

For an example of modeling with the the M^{3}L, consider the definition of a conditional statement found in imperative programming languages in Figure 3. It consists of *Condition* to decide whether to execute *ThenStatement* or *ElseStatement*.

Semantic rules can be defined on concepts, denoted by "|=". Figure 2e shows the graphical notation. A semantic rule references another concept that is returned when a concept with a semantic rule is referenced. As with any other reference, a non-existent concept is created on demand.

Context, specializations, and semantic rules are employed for *concept evaluation*. A concept evaluates to the result of its syntactic rule, if defined, otherwise itself, . Syntactic rules are inherited from explicit base concepts (given by *is a* or *is the*) and implicit base concepts (concepts with matching content).

Concept evaluation is used to assign semantics to concepts. The code in Figure 4 uses syntactic rules to assign semantics to the conditional statement from the example above. A concrete statement is matched against the two subconcepts *IfTrueStmt* and *IfFalseStmt*. If one of them is recognized as a derived base concept of the given statement, the semantic rule of the matching concept is inherited. This way, the *ThenStatement* or the *ElseStatement* is determined as the evaluation result of a *ConditionalStatement* and thus a specialization of the respective statement is "executed" (evaluated next).

```
Java is a ProgrammingLanguage {
  ConditionalStatement
    ⊢ if ( Condition ) ThenStatement
      else " " ElseStatement .
}

Python is a ProgrammingLanguage {
  ConditionalStatement
    ⊢ if " " Condition :
      "\n  " ThenStatement
      else:
      "\n  " ElseStatement .
}
```

Figure 5. Sample syntax of the conditional statement.

Concepts can be marshaled/unmarshaled as text by *syntactic rules*, denoted by "|-" or graphically as shown in Figure 2f. A syntactic rule names a sequence of concepts whose representations are concatenated. A concept without a syntactic rule is represented by its name. Syntactic rules are used to represent a concept as a string as well as to create a concept from a string.

Figure 5 shows syntactic rules that map the conditional statement from the example to different programming languages. Chosing the programming language as the context, a syntactic form of the concepts is generated accordingly.

B. MDSE with the M^{3}L

An MDSE process relies on a series of models where models are created from existing models by means of model-to-model transformations. A model on one stage is created based on the input of models of earlier stages or by refining models from the same stage. This article considers three typical kinds of model transformations:

1) Model combination
2) Model refinement
3) Model creation from existing models

The three model relationships can be used with the M^{3}L to express model transformations. The following examples outline basic modeling approaches.

a) Combining models: Domains often rely on base domains. For example, business tasks rely on mathematical models. Accordingly, models are defined by integrating (existing) models of base domains. This way, models are reused.

Let *BaseModel1* and *BaseModel2* be some models of some domains whose concepts can be reused for the domain at hand. Then, for example, concepts *A* and *B* can be integrated into a new model *SomeModel* by definitions like:

```
SomeModel {
  A from BaseModel1
  B from BaseModel2
}
```

For example, on the layer of domain models, the model shown in Figure 6a combines parts of product details that come

```
ProductDescriptions is a DomainModel {
  ProductData
  PaymentMethods        from Commerce
  PackagingInformation from Logistics
}
```

(a) A sample domain model.

```
OurInfoSys is a PlatformIndependentModel {
  AppServer  from SWComponents
  DBMS       from SWComponents
  DataSchema from DBModeling
  WebServer  from SWComponents
  WebPage    from WebDesign
}
```

(b) A sample solution architecture.

```
OurInfoSysConcept is an OurInfoSys {
  RDBMS from SWComponents is the DBMS
  ProductDataSchema
    is an  RDBSchema from DBModeling,
       the DataSchema
  WebServer from SWComponents
    is a ServletEngine from Java
}
```

(c) A sample solution architecture refinement.

```
OurInfoSysConcept ⊨ OurInfoSysDataLayer {
 RDBMS
 ProductDataSchema {
  ProductsTable is a Table from DBModeling
 }
}
```

(d) A sample software architecture.

```
OurInfoSysDBIm is an OurInfoSysDataLayer {
 ProductDataSchema {
  ProductsTable ⊢ "PRODUCTS(" Columns ")".
 } ⊢ "CREATE TABLE " ProductsTable .
}
```

(e) A sample software architecture.

Figure 6. Sample flow of software models expressed in the M^{3}L.

from different specialized models (assuming that concepts for models *Commerce* and *Logistics* are given, and that those models combine the named concepts for the data sets).

Likewise, on the layer of solution architecture, the model from Figure 6b combines technical components from different technical descriptions. Here, we assume the availability of a model *SWComponents* that hosts descriptions of typical software components found in the domain at hand, etc.

b) Refining models: Within one stage, models are refined to more concrete models of the same stage. This way, the work in each stage starts with first, coarse-grained models, that are then transformed into more concrete models. Different refinements of one model may cover different perspectives on the targeted (software) solution. The process of refining involves decision making. Decisions can be documented by stating delta models that explicitly represent the transformation applied during refinement.

Using the M^{3}L, one model can be created as a refinement of another. Concepts in the content of the refined model are inherited and can be refined further.

An example from the solution architecture layer is shown in Figure 6c. In this example, two aspects of the conceptual model are refined: From a technical perspective, the *DBMS* is more concretely specified to be a relational DBMS (*RDBMS*), and the *WebServer* to be implemented as a Java Servlet engine (*ServletEngine*). Regarding the domain model, it is defined that the data schema is defined to store products (*ProductDataSchema*).

c) Creating models in subsequent stage: When processing from one stage to another, initial models are required for the subsequent stage that is entered. Optimally, the most concrete models of the preceding can be transformed to form the initial models of the subsequent stage. If new models have to be created, the model elements should be explicitly linked to the elements from models on which they are based for the sake of traceability. For example, Shaw and Garlan [14] demand for software architecture that solution aspects refer to requirements.

In the M^{3}L, a model can be explicitly created as a transformation of another model using a semantic rule.

Figure 6d continues the example of the information system. *RDMBS* from the source model *OurInfoSysConcept* is reintroduced in the transformed model. The database schema *ProductDataSchema* is additionally redefined by naming one table. *WebServer* from *OurInfoSysConcept* is not considered in the transformed model, since it only models the data layer of the information system.

C. Software Creation with the M^{3}L

The models in MDSE ultimately reach the stage of generating code. The M^{3}L allows creating code using syntactical rules that can be added to models with sufficient concreteness.

Using the example from above, part of the information system based on a relational database can be defined to create a relational schema by SQL statements as indicated in Figure 6e.

By defining the syntactical rules in the context of an implementation model, different code generation schemes can be defined for one software model. This way, for instance application code, UI code, data models, and data exchange formats can be generated from the same model.

V. First Experiments Using the M^{3}L

Describing static documents with metadata provided as concepts that make reference to relevant parts of the content has been researched previously in the *Concept-Oriented Content Management* approach [15]. The M^{3}L may offer new way to link abstract concepts with documents. Some initial experiments with simple mutable documents have been conducted to investigate means of linguistic document interpretation.

A. Static Document References

Figure 7 is an example of a document description using the M^{3}L. It illustrates static references to documents or document fragments. It uses an example from art history as the primary application domain used in the initial studies of document descriptions in the Concept-Oriented Content Management approach. Applications from the humanities were chosen because of the specific need for multifaceted interpretations of documents. Many of the findings also apply to the software domain since the requirements are similar, if not more stringent.

The (digital) picture of a painting at the bottom of Figure 7 is described using (M^{3}L) concepts. An abstract concept hierarchy, beginning with the concept *DocumentReference*, defines references to documents or document fragments. A *DocumentId* defines an address of a document, such as a file name or URL. A *FragmentSelector* defines a part of a document containing interesting content. The example shows a sketch of a refinement hierarchy that specifies concepts for references to two-dimensional images, for those depicting paintings, and paintings showing a ruler specifically.

A second concept hierarchy starts with *DocumentDescription* and contains concepts that describe the subject of a document. The concepts in this hierarchy describe an interpretation of a painting, its content. There is little to no abstraction in general. Instead, the concepts lay a foundation for individual concepts that record observations and interpretations.

The two hierarchies converge at the *PaintingDescription*. The application-specific concept *RulerPaintingDescription* refines it for the area of interest. In this case, it is ruler images that convey a political message. Therefore, there is a reference to a *Ruler* concept describing the ruler that the artwork is about and a fragment selector *RulerDepiction* pointing to the ruler's depiction in the picture. Additional content can be added as needed, such as a description of the artist, the epoch, the creation location, or the exhibitions of the artwork.

Finally, *NapoleonCrossesTheAlps* provides an "instance" of a ruler painting. There are actually multiple paintings of that motive, and even more references to Hannibal in depictions of Alp crossings. The concept *NapoleonCrossesTheAlps* unites multiple views of the (historical) content and the document (painting). It also relates multiple domains, such as history, arts, and political science.

B. Interpretation of Semistructured Documents

Some general concepts serve as the foundation of the interpretation of certain types of documents. These concepts are used to assign domain semantics to content.

Figure 8 shows an example of documents representing customer journeys that are drawn on and exported from (hypothetical) whiteboard software. This article assumes a hypothetical service because the APIs for accessing board content from actual services differ. However, they typically allow access to graphical shapes. Examples include the APIs of the services Miro [16] and FigJam [17].

The concepts shown Figure 8 in facilitate the interpretation of the graphical components in the UX design domain. Specializations of the concept *Board* allow referencing a whiteboard, and specializations of the concept *Page* allow referencing a page (assuming the whiteboard software allows whiteboards to be subdivided). On a whiteboard, there is no recognizable structure below the page level. Starting with the concept *CustomerJourney*, we look for semantic structures on a whiteboard page. A customer journey is a named (graphical) object that consists of elements that represent *Touchpoint*s and ones that represent *Step*s. A touchpoint is characterized by a *Name* and a *Service*.

Syntactic rules define how these concepts are represented on a whiteboard page. Figure 8a sketches some rules that generate and recognize JSON code as it may be exported from a whiteboard software that is provided as a Cloud service.

Service and *Touchpoint* are each represented by one shape. These shapes, and thus these two entities, are reflected in the whiteboard service API. However, compound entities have no counterpart in the API; *CustomerJourney* specializations are defined in M^{3}L concept structures.

Once a customer journey has been developed on a whiteboard of that form, the syntactic rules can be applied to recognize its structure and to extract its content. Content changes, such as renaming an entity, lead to updated concepts. New concepts are created for elements that are added to the board, and they will be added to the customer journey corresponding to a board page.

Figure 8b shows a sample query for customer journeys. The concept for the board is a subconcept of *Board* from Figure 8a. It matches all board specializations that refer to the given board (i.e., the *URL*), have the name *CustomerJourneys*, and contain a page called *CustomerJourney3*. Furthermore, the page must contain a *CustomerJourney* that has one *TouchPoint* with the *Website* service.

If a board matching the query exists, it will be selected. Otherwise, it will be updated accordingly.

When the complete board is interpreted according to the syntactic rules for *Board*s, the result is the concept structure shown in Figure 8c. The concepts that have been created from the board reflect some of the design decisions from the customer journey representation, such as the participating persona and the relationships to the touchpoints it visits and the sequence of touchpoints along the customer journey.

The extracted information can be used in subsequent activities of the software development process. Using the M^{3}L, the resulting concepts can be directly related to concepts that represent models created in such subsequent activities. With the relationships outlined in Section IV-B, the related software models are updated on changes to the board.

C. Reinterpretation of Mutable Documents

Mutable documents are handled by repeatedly applying the parsing process. When reinterpreting a document after a change, new concept definitions are created in the M^{3}L. Because the M^{3}L matches definitions against existing concepts before creating new ones, previous interpretations are found and used in the parsing process. Depending on the concept model,

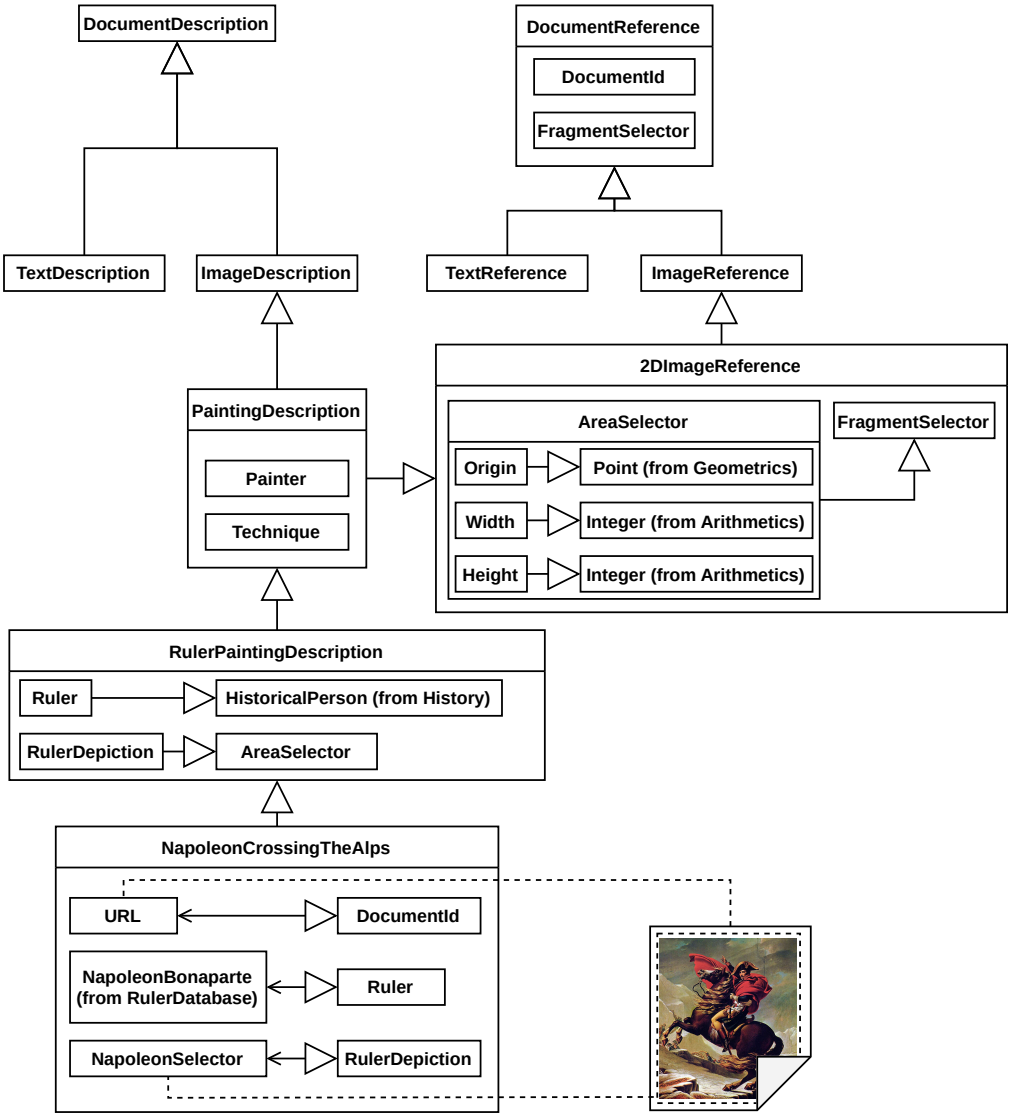


Figure 7. Static references to documents and document fragments.

existing references to concepts in models from subsequent stages that were established by model-to-model transformations are preserved. In this way, documents can be modified even after they have already been interpreted and related to other models during an MDSE process.

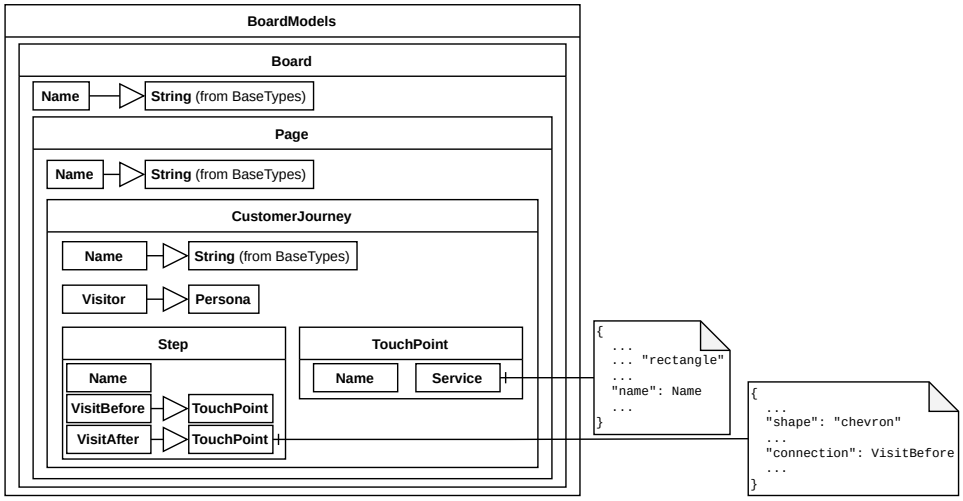
Structural changes can be recognized to a limited extent. As indicated in the previous subsection, newly added shapes lead to newly created concepts. But the correct linking of the concepts, such as the order given by *VisitBefore* and *VisitAfter*, cannot be established using the sample model shown so far. In the example of the customer journey visualization, the order of the service usages might be visualized by the horizontal position of the rectangle shapes. Interpretation requires topological relationships "left of" or "right of", or the horizontal positions of the shapes have to be interpreted in a pragmatic way.

Alternatively, the order of touchpoint visits may be visualized explicitly by arrows as indicated in Figure 8b. Such a visualization may be harder to analyze, depending on the way arrows are reflected in the board software's API. Since arrows are expected to be manually drawn in most cases, they will not be represented by curves or splines, making it very hard to analyze them with syntactical rules.

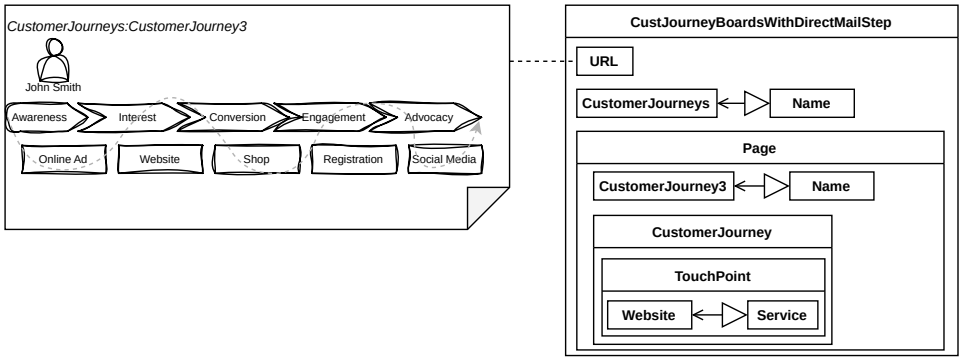
Recognition of existing concepts requires some stable identity information. Otherwise, the associative matching as provided in the M^3L might fail to select the right concepts. Such stable identity information may, for example, be unique names as well as a certain location in the document structure where it is placed. In the example of the digital whiteboards above, names might be given in a specially positioned text field. As a consequence, the documents are not completely mutable, at least not in terms of content.

An agreement on some recognizable information constitutes a restriction to the idea of mutable documents. Finding ways of leveraging this situation is subject to future work.

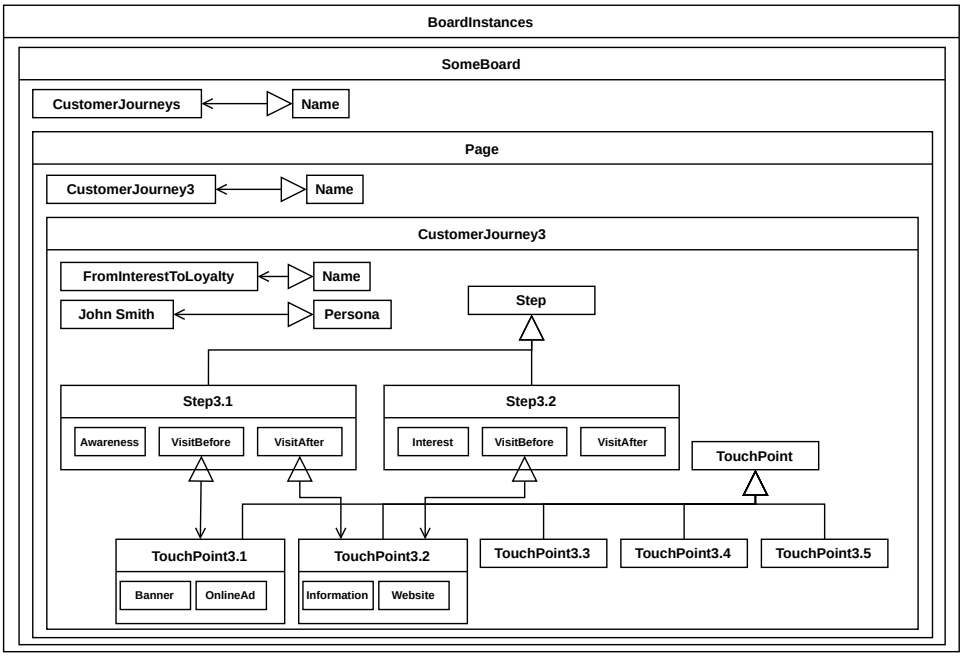
Other structural changes require extensions of the meta model of the current modeling stage. If, for example, a new



(a) Example of a pattern for mutable documents.



(b) Example of a query to mutable documents.



(c) Example result of mutable document recognition.

Figure 8. Parsing of documents and document fragments.

kind of entity is introduced, and if that entity is represented by a new shape in whiteboard drawings, then a new concept for the entity with a syntax rule for its representation has to be added.

By means of model compositions (see Section IV-B), though, models with matching syntax definitions might be looked up from a model repository. Such dynamic extensions are also subject to future research.

VI. Conclusion and Future Work

In this article, we investigate an approach to integrate semi-structured documents supporting creative activities into MDSE processes. Using the M^3L, documents can be parsed based on their syntactic structure in conjunction with the semantics of the concepts represented in such documents. A first simple experiment shows that content can be extracted from a document in a suitably formal form if the document follows some conventions. The concepts recognized in a document can serve as model elements that link the documents to the chain of model-to-model transformations of MDSE processes.

Future work will need to test this approach with a range of existing file formats and service APIs to further investigate the limits of document interpretation and possibly identify additional requirements for parsing technology. There are limits to the extent to which documents can be modified without losing existing links to software models. These limits are not well researched. We need to find the limits, ways to extend them, and notations to describe parts of documents that must not be altered. Another future research direction concerns a form of roundtrip engineering in which documents are not only interpreted, but also generated from models that need to be presented in a form suitable for non-technical stakeholders.

GenAI receives tremendous attention lately. Though not part of the original research, it has to be included in future research. GenAI can support multiple modeling activities, in particular in conjunction with creative artifacts.

Some GenAI applications have already been researched, such as checking completeness of requirements (given in natural language) [18]. After the emerge of "vibe coding" [19], there is an interest in "vibe modeling" as an iterative model transformation approach. It may also help in generating code from models.

Currently, we are looking in low code tools with generic AI support. In the future, it will be interesting to see whether LLMS that have been trained specificially for modeling with the M^3L allow automated model transformations by stating a source model and giving delta model instructions.

Acknowledgement

The author thanks the Nordakademie for granting the opportunity to publish this work.

References

[1] H.-W. Sehring, "Integrating creative artifacts into software engineering processes", in *Proceedings of the Seventeenth International Conference on Creative Content Technologies*, ThinkMind, 2025, pp. 1–6.

[2] D. Schmidt, "Guest editor's introduction: Model-driven engineering", *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[3] G. Liebel et al., "Human factors in model-driven engineering: Future research goals and initiatives for mde", *Software and Systems Modeling*, vol. 23, no. 4, pp. 801–819, 2024.

[4] E. Herac, L. Marchezan, W. Assunção, R. Haas, and A. Egyed, "A flexible operation-based infrastructure for collaborative model-driven engineering", in *Modellierung 2024*, ser. Lecture Notes in Informatics, Gesellschaft für Informatik e.V., 2024.

[5] I. Galvao and A. Goknil, "Survey of traceability approaches in model-driven engineering", in *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, 2007, pp. 313–313.

[6] H.-W. Sehring, "Visual artifacts in software engineering processes", in *Proceedings of the Sixteenth International Conference on Creative Content Technologies*, ThinkMind, 2024, pp. 1–6.

[7] S. Trujillo, M. Azanza, and O. Diaz, "Generative metaprogramming", in *Proceedings of the 6th international conference on Generative programming and component engineering GPCE '07*, Association for Computing Machinery, 2007, pp. 105–114.

[8] J. Arnoldus, M. Van den Brand, A. Serebrenik, and J. J. Brunekreef, *Code generation with templates*. Springer Science & Business Media, 2012, vol. 1.

[9] K. Lano and Q. Xue, "Code generation by example using symbolic machine learning", *SN Computer Science*, vol. 4, Jan. 2023.

[10] W. Ding, X. Lin, and M. Zarro, *Information Architecture and UX Design: The Integration of Information Spaces*. Springer Cham, 2025.

[11] H.-W. Sehring, "Model-supported software creation: Towards holistic model-driven software engineering", in *Proceedings of the 2023 IARIA Annual Congress on Frontiers in Science, Technology, Services, and Applications*, ThinkMind, 2023, pp. 113–118.

[12] I. Amous, A. Jedidi, and F. Sèdes, "A contribution to multimedia document modeling and querying", *Multimedia Tools and Applications*, vol. 25, pp. 391–404, 3 Oct. 2005.

[13] A. Roy, K. Ghosh, M. Basu, P. Gupta, and S. Ghosh, "Retrieving information from multiple sources", in *Companion Proceedings of The Web Conference 2018*, International World Wide Web Conferences Steering Committee, 2018, pp. 43–44.

[14] M. Shaw and D. Garlan, "Formulations and formalisms in software architecture", in *Computer Science Today: Recent Trends and Developments* (Lecture Notes in Computer Science), Lecture Notes in Computer Science. Springer, 1995, vol. 1000, pp. 307–323.

[15] J. W. Schmidt and H.-W. Sehring, "Conceptual content modeling and management", in *Perspectives of System Informatics*, Springer, 2003, pp. 469–493.

[16] Miro, "Get specific item on board", 2025. [Online]. Available: https://developers.miro.com/reference/get-specific-item.

[17] Figma, "Node types", 2025. [Online]. Available: https://www.figma.com/plugin-docs/api/nodes.

[18] D. Luitel, S. Hassani, and M. Sabetzadeh, "Improving requirements completeness: Automated assistance through large language models", *Requirements Engineering*, vol. 29, no. 1, pp. 73–95, 2024.

[19] A. Gadde, "Democratizing software engineering through generative ai and vibe coding: The evolution of no-code development", *Journal of Computer Science and Technology Studies*, vol. 7, no. 4, pp. 556–572, 2025.