A Domain-Specific Language For Specifying Requirement Patterns for Model-Driven Software Development

1st David Mosquera
School of Engineering,
Institute, of Computer Sciences
ZHAW Zurich University of Applied Sciences
Winterthur, Switzerland
UPV Universitat Politècnica de Valècnia
Valencia, Spain
mosq@zhaw.ch

2nd Marcela Ruiz
School of Engineering,
Institute of Computer Sciences
ZHAW Zurich University of Applied Sciences
Winterthur, Switzerland
ruiz@zhaw.ch

3rd Anastassios Martakos
School of Engineering,
Institute of Computer Sciences
ZHAW Zurich University of Applied Sciences
Winterthur, Switzerland
anastassios.martakos@outlook.com

***Abstract*—Requirement patterns have emerged in requirements engineering as a way to streamline requirements specification by promoting reuse. However, existing approaches in the literature have primarily focused on defining requirement patterns for reuse within textual requirement documents, overlooking software models. Meanwhile, other model-driven software development (MDSD) approaches that attempt to reuse software models as patterns for requirements specification are tied to specific modelling languages, limiting their applicability. In this paper, we introduce the LEMON specification language: a domain-specific language designed to specify and reuse requirement patterns for MDSD tools. The LEMON language supports the specification of patterns, templates, and the integration in any MDSD tools' metamodels, allowing requirements patterns customisation and their integration in MDSD environments. We present an Extended Backus-Naur Form (EBNF) of the LEMON language exemplified by means of an industry-inspired example. Moreover, we describe the implementation of an execution environment to show the integration of the LEMON language in an MDSD tool. We conclude with our vision and outline future steps for the evolution of the LEMON language to streamline requirements specification in MDSD tools using requirement patterns.**

Keywords—Requirement Patterns, LEMON, Model-driven software development, Requirement specification

I. Introduction

In requirements engineering, the elicitation and specification of software requirements are essential but often time-consuming and error-prone activities, resulting in incomplete and missing software requirements [1]. To address these challenges, requirement patterns have been proposed as reusable solutions for recurring stakeholder needs [2], [3]. These patterns aim to improve consistency, reduce specification effort, and promote best practices by capturing validated structures for both functional and non-functional requirements.

Some authors [4], [5], [6], [7], [8], [9], [10] have proposed approaches for reusing requirement patterns for specifying requirements as textual requirement documents, limiting its applicability in MDSD as software models are the main artefact instead of text descriptions. Other authors [11], [12], [13], [14], [15], [16] have proposed approaches to reuse requirement patterns to generate software models for MDSD. However, these approaches are tied to specific modelling languages, MDSD tools, or domains. In our previous work, we introduced the LEMON framework [15], supporting the reuse of requirement patterns in MDSD through black-boxed modules and assistant-based integration. However, this prior work did not define how to specify the requirement patterns and how to execute them.

Thus, in this paper we present the LEMON language—a domain-specific language (DSL) for specifying reusable requirement patterns in a tool-agnostic, model-oriented way. The LEMON language allows requirements engineers to define patterns independently of a particular MDSD tool but still capable of being integrated with metamodels and instantiated within MDSD tools. The language builds on formal specification principles from related works and includes grammar-based constructs for defining metamodels, patterns, templates, and implementations. We describe the LEMON language syntax and semantics using the Extended Backus-Naur Form (EBNF). Moreover, we outline the architecture of its execution environment, which includes a language editor, a pattern catalogue, and an interpreter for generating MDSD tool models, supporting the lifecycle of requirement pattern reuse—from specification to instantiation—while enabling integration into existing MDSD tools.

While the LEMON language and its execution environment present a promising foundation, further research is needed to evolve them from a proof of concept to a fully mature solution. To this end, we conclude the paper by outlining future development steps and formulating key research questions that define the ongoing research agenda for the LEMON language.

This paper is structured as follows: in Section II, we present the related works; in Section III, we present a running example to motivate and exemplify our approach; in Section IV, we describe in detail the LEMON language EBNF; in Section V, we introduce the execution environment that supports the language; and, finally, in Section VI, we draw conclusions and future work.

II. Related Works

Requirement patterns approaches have been proposed in literature. Methods, catalogues, and languages, among other approaches have been envisioned. For the sake of this paper, we classify existing work into two main categories:

- Approaches that aim to reuse requirement patterns to generate text-based requirement documents [4], [5], [6], [7], [8], [9], [10].
- Approaches that aim to reuse requirement patterns to generate software models for MDSD [11], [12], [13], [14], [15], [16].

The aim of the first type of approaches is to generate requirement documents, selecting from a catalogue of requirement pattern templates or controlled natural language, producing a new requirement document into a requirement book.

In this category, we observe proposals as PABRE (Pattern-Based Requirements Elicitation; [4], [9]) and CaRePa (Context-aware Requirement Patterns; [5]) that demonstrate how template-based requirement patterns can guide the specification process by facilitating the creation of new requirements as documents. Similarly, Darif et al. [10]ntroduced the Unified Template Language (UTL), which offers tool support for the specification of requirements using controlled natural language, particularly to create requirements based for safety-critical systems. Other authors as Sardi et al. [6] and Sleimi et al. [7] have proposed domain-specific text-based catalogues for requirement pattern reuse. Finally, Denger et al. [8] use MDSD principles to propose a controlled natural language for embedded systems requirement pattern specification, to reduce the imprecision in natural language requirements specifications with the use of natural language patterns.

While these approaches provide valuable support for text-based elicitation and specification, they often overlook the model-driven perspective, where requirements are expressed directly as software models within Model-Driven Software Development (MDSD) tools. MDSD emphasizes the use of abstract models as the main artefacts for system specification that can be automatically transformed into the software that meet stakeholders' requirements [17].

Second category of related works, arise as a way to leverage requirement patterns to generate and reuse software models in MDSD instead of text-based documents. The approach we present in this paper (i.e., the LEMON language) falls in this category.

In this category, we observe proposals from Robertson [11] and Silva et al. [12] that show how event and use case diagrams can be systematically used to extract recurring requirements and organize them into reusable model-based catalogues. Some authors as well have applied UTL to automatically generate user interface models [16]. In practice, MDSD tool providers have adopted ad-hoc requirement patterns catalogues to reuse software models from specific tools, resulting in solutions as the Mendix Sample and Starter App Catalogue [13] and OutSystems Application Templates (or Application Forge) [14].

The approaches are capable of reusing software models but are limited by the syntactic and semantic constraints of specific modelling languages and MDSD tools. As a result, the requirement pattern specifications lack flexibility for cross-tool integration and do not fully support conceptual abstraction across heterogeneous platforms and domains. Thus, in this paper, we aim to bridge this gap proposing the LEMON language: a DSL for specifying reusable requirement patterns in a tool-agnostic, model-oriented way for requirement specification in MDSD.

The LEMON language builds upon our previous work, in which we introduced the LEMON framework [15]. The LEMON framework consists of black-boxed modules where patterns need to be specified and stored into a catalogue, that later a requirements engineer can reuse and integrate into an MDSD tool using an assistant. However, our earlier work presented the framework only at a high level, without detailing the design of the underlying specification language needed to create and reuse the requirement patterns. In this paper, we address this gap by presenting the design of the LEMON language, along with illustrative examples, and a proof of concept for requirement pattern edition and catalogue construction.

III. Running Example: The case of Whatscount[1]

Whatscount GmbH is a young Swiss software development company specializing in digital transformation. The company operates in various domains, including digital health, Industry 4.0, and finance. Whatscount develops its software solutions using the MDSD tool named Posity Design Studio (PDS) [18]. PDS enables the creation of data-centric applications through the use of six types of models. In this paper, we focus on two foundational models used in PDS: the table model and the query model. These models serve as the basis for the subsequent construction of user interface models, process models, and role access models.

In practice, Whatscount observed a recurring requirement from stakeholders: the need to visualize data, apply filters, and search for specific records. This led to the repeated creation of query models grounded in predefined table models. Over time, this activity emerged as a common and reusable solution to a frequent problem—a candidate for a requirement pattern.

Whatscount's requirements engineers now seek a way to document and reuse this solution while eliciting requirements from their stakeholders. They aim to abstract this recurring requirement into a pattern that can be systematically applied across projects. This raises a central question:

[1] Although the LEMON Language is demonstrated using a specific MDSD tool (PDS) from our running example, the LEMON language itself is designed to be tool-agnostic. The language's constructs for metamodels and implementations allow integration with arbitrary tools, as long as their structural metamodels are defined in LEMON. We acknowledge current implementation, and examples are tied to PDS as it serves here as an industry-inspired example, but future work will extend support to other MDSD tools to test its flexibility.

How can a domain-specific language help Whatscount specify a reusable requirement pattern that aligns with their MDSD tool, PDS, and supports model-level reuse? In Section IV we present the design of the LEMON language as a solution to this research question.

IV. Design: The LEMON Specification Language

The LEMON specification language focuses on specifying requirement patterns in model-driven software development tools. The LEMON language consists of four main elements: i) patterns; ii) templates; iii) implementations; and iv) metamodels. These elements are used to specify requirement patterns and templates (see Section IV.A), to later implement the templates into a metamodel (see Section IV.B). We show the LEMON language metamodel in Fig. 1.

To formally define the statements in the LEMON language, in the following subsections, we provide a subset of the language grammar using an Extended Backus-Naur Form (EBNF) notation [19].

A. *Patterns & Templates Specification*

The main purpose of the LEMON language is to specify requirement patterns. We design the LEMON language elements for pattern specification relying on PABRE [4]. Thus, we introduce the first language element: **pattern.**

A **pattern** is a named, self-contained element that represents an abstract description of a reusable requirement pattern, functional or non-functional. Each pattern encapsulates its core purpose independently of any specific implementation or MDSD tool, serving as a conceptual and technology-agnostic specification. This specification is relevant as it will help requirements engineers to decide whether the specified requirement pattern is applicable to the project at hand or not, as well as later creating **templates** that implement the pattern into specific MDSD tools. It includes a name, a textual description, a defined goal, an author, and a set of keywords. Additional metadata may also be included to provide further context and facilitate later retrieval (e.g., sources, images, links, comments, and classification schemas). To express the **pattern** definition formally, we define the LEMON language syntax using the EBNF in the following Listing. We provide an example of a pattern definition named "Searchable and Editable List" whit ID "CreateList" in the context of Whatscount in Annex VII.A.

Listing: Pattern_definition

```
pattern id = ID
 metadata
  name name_text_term = Term
  description description_text_term = Term
  goal goal_text_term = Term
  author author_text_term = Term
  keywords keywords_text_term += Term (',' Term)*
  additional_metadata += (Additional_metadata)*
 endmetadata
endpattern
```

Requirement patterns can be specialized into **templates** (a.k.a. forms). A **template** aims to achieve the same goal as the requirement pattern from which they are specialised, but with different levels of granularity, parameters, and implementing the pattern into one or more MDSD tools. Forms have a name, description, author, version, and extra metadata, as well as **fixed** and **extended parts.** In the LEMON language, however, instead of creating text as other approaches, reusing a **template** will create software models in an MDSD tool, using the

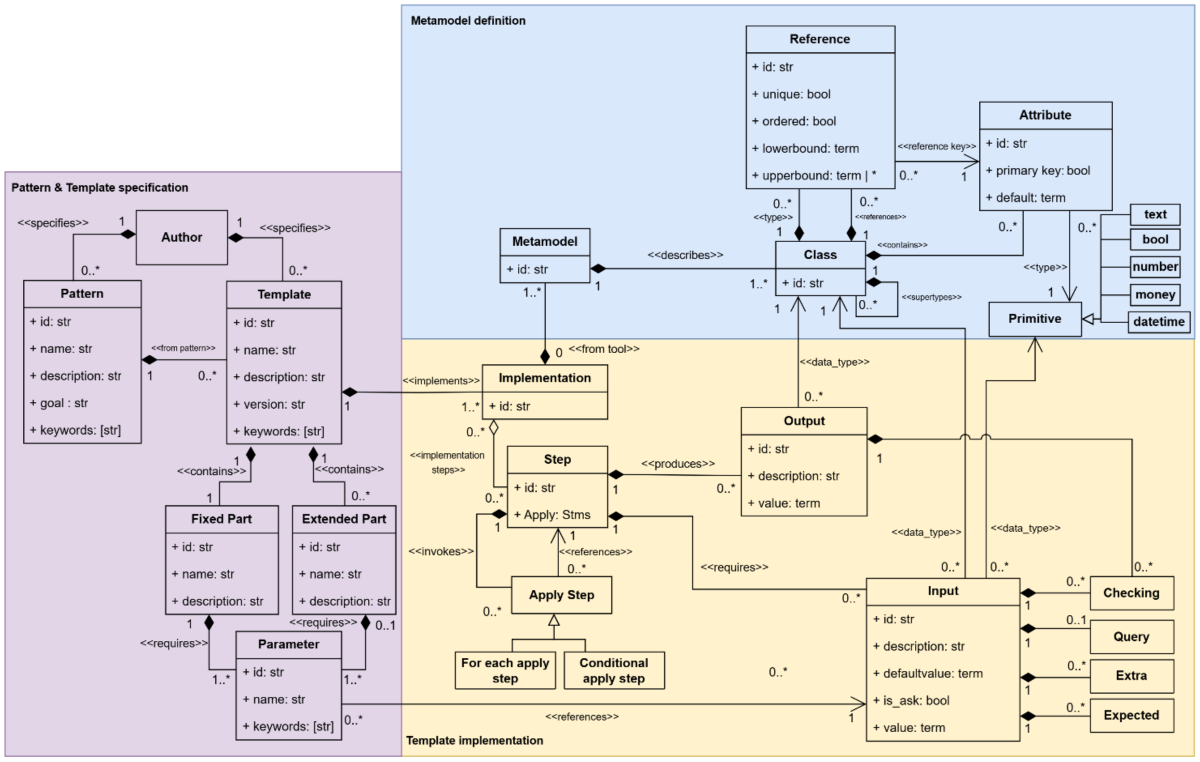


Fig. 1. LEMON Language Metamodel.

implementation element. To express the **template** definition formally, in the following paragraphs we show the LEMON language syntax using the EBNF, starting with **template** definition.

Listing: Template_definition

```
template id = ID from pattern
                    pattern_ref = [Pattern_def]
 name name_text_term = Term
 description description_text_term = Term
 author author_text_term = Term
 keywords keywords_text_term += Term (',' Term)*
 version version_text_term = Term
 additional_metadata += (Additional_metadata)*
 fixed_part = Fixed_part_def
 (extended_part += Extended_part_def)*
  Implementations += Implementation_def*
endtemplate
```

Fixed parts are mandatory when defining the **template**, having only one per **template**. Parts describe what is needed to achieve the requirement pattern goal without describing how to achieve it, following what is defined in PABRE [4]. It contains a name, a description, and a set of **parameters**.

Listing: Fixed_part_definition

```
fixedpart id = ID
 name name_text_term = Term
 description description_text_term = Term
 part_parameter_stmts += Parameter_def*
endfixedpart
```

Parameters are the **inputs** to be instantiated in order to **implement** a **template** into a specific MDSD tool. **Parameters** include a description/question to be used when reusing the **template**, and a set of keywords as additional semantical markup.

Listing: Parameter_definition

```
parameter
  input_name_reference =
  [Input_stmt | Reference_Name]
  description_text_term = Term
  as additional_semantical_markup_keywords +=
  Term (',' Term)*
```

Extended parts are similar to fixed parts, containing extra **parameters** to achieve the goal of the requirement **pattern** for specific domains, tools, or constraints. They share the same structure as **fixed parts**.

Listing: Extended_part_definition

```
extendedpart id = ID
 name name_text_term = Term
 description description_text_term = Term
 part_parameter_stmts += Parameter_def*
endfixedpart
```

We show an example implementing a **template** named "List and Search in PDS" that extends the requirement **pattern** "CreateList" for re-using it in PDS in Annex VII.B, including **fixed parts** for setting up a generic query model, including **parameters** for the query's name and its primary data source. As well, it includes an **extended part** that specializes the **template** for digital health applications, guiding the configuration of software models that visualize test data such as vital signs or lab results.

B. *Metamodel Definition & Template Implementation*

After specifying the **patterns** and **templates** as shown in Section IV.A, the LEMON language bridges the gap from requirement patterns to MDSD tools through the **metamodel definition** (see Section IV.B.1) and **template implementation** (see Section IV.B.2).

1) *Metamodel Definition*

MDSD tools have an underlying metamodel. Metamodels define how model elements are related and interconnected structurally to each other. To allow the LEMON language to specify how a **template** is **implemented** into a specific MDSD tool, a **metamodel** needs to be defined.

A **metamodel** is a named element in the LEMON language that groups the set of **classes** from the MDSD tool to be instantiated. We design the **metamodel** element by reusing a sub set of Ecore [20] metamodel elements (i.e., EClass, EAttribute, and EReference). To express the **metamodel** element formally, in the following paragraphs we show the LEMON language syntax using the EBNF, starting with **metamodel** definition.

Listing: Metamodel definition

```
metamodel id = ID
  metamodel_definition_stmts += Metamodel_class_def
endmetamodel
```

Classes are named elements used to describe models (equivalent to EClass in Ecore). Classes contain attributes and references, and they can have superclasses for inheritance.

Listing: Metamodel_class_definition

```
class id = ID
(supertypes += [Metamodel_class_def]+)?
  metamodel_class_def_stmts +=
  (Class_attribute_def | Class_reference_def)*
endclass
```

Attributes are typed elements of a **class** that hold a primitive value (e.g., text, int). It can have a default value, as well as being the primary key of the model. Equivalent to EAttribute in Ecore.

Listing: Class_attribute_definition

```
attribute id = ID
 type data_type = number | text | bool …
 default term = Term
 primarykey primary_key_bool = Condition_term
endattribute
```

References, on the other hand, are elements of a **class** that represent a relationship to another **class.** It can be unique, define if it is ordered or not, the keys, the lower bound and upper bound of the reference. Equivalent to EReference in Ecore.

Listing: Class_reference_definition

```
reference id = ID
 type class_name_reference = [Metamodel_class_def]
 ordered ordered_bool_term = Term
 unique unique_bool_term = Term
 keys class_name_attribute_name_references+=
[Class_attribute_def | ReferenceName])+
 lowerbound lower_bound_number_term = Term
 upperbound upper_bound_number_term = Term | *
endreference
```

Using **classes, attributes, and references** Whatscount can define the PDS **metamodel**, defining classes such as Table and QueryModel. We show an example of this definition in Annex VII.C.

2) Template Implementation

Patterns and **templates** elements describe the abstract yet conceptual part of the requirement patterns. **Metamodels** define the **classes** that can be instantiated into a specific MDSD tool. **Template implementation,** then combine the **templates** and **metamodels** to reuse a requirement pattern into an MDSD tool.

The main element is the **implementation,** a named element that describes how to achieve the requirement **pattern** goal from a specific **template** by transforming a set of **inputs** into **outputs** from a **metamodel** (or set of metamodels) by **applying** a set of **steps**. To express the **implementation** element formally, in the following paragraphs we show the LEMON language syntax using the EBNF, starting with **implementation** definition and followed by **inputs, outputs,** and **apply steps.**

Listing: Implementation_definition

```
implementation id = ID
 tools metamodel_ref =
 [Metamodel_def | Referencename]
 (',' [Metamodel_def | ReferenceName])*

 implementation_stmts +=
 (Input_def | Output_def | Apply_step )+

endimplementation
```

Inputs are named elements in the LEMON language that define the data required to execute a **template implementation**. They can be: a single input or a list of inputs, and either primitive types (e.g., text, number, boolean) or instances of classes from the referenced metamodels. Each **input** may have a predefined value or be marked as ***ask***, meaning its value must be provided interactively during **template** reuse using a **query**. In addition, inputs can include **checks** to validate specific logic, **expected** attributes to enforce mandatory values, and **extras**—extended attributes not originally defined in the metamodel but required for the **template implementation**. To express the **input** element formally, in the following paragraphs we show the LEMON language syntax using the EBNF, starting with **input** definition and followed by each sub statement.

Listing: Input_definition

```
(input | inputlist) id = ID
   type data_type =
   (number | text | bool …) |
   ([Class_reference_def | ReferenceName])
  description description_text_term = Term
  (value | defaultvalue) value_term = Term | ask
  (query_def = Query_def)?
  additional_input_def +=
  (Extra_def | Expected_def | Checking_def )*
(endinput | endinputlist)
```

- *Single Input:* Represents an **input** that holds only one value, which may be null, a primitive, or a class instance.
- *Input List*: Represents an **input** that contains a list of values, which may be empty or composed of primitive values or class instances.
- *Primitive Input*: Indicates that the **input** type is a primitive, such as text, number, or boolean.
- *Class Input:* Indicates that the **input** type is an instance of a **class** defined in the referenced **metamodel**—conceptually equivalent to an object in object-oriented programming.
- *Default Value:* A constant value assigned to the **input**. It does not change and is always resolved to the default when the **template** is reused.
- *Ask Value*: A dynamic value provided at the time of **template** reuse. ***Ask*** values are mutable and referenced by **parameters** in the **fixed** or **extended parts**.
- *Query for Ask Value from Class Input*: When an ***ask*** value is defined over a **class input**, it may include a **query** that retrieves candidate instances from the MDSD tool. Queries are defined using a SQL-like syntax that allows selecting from **class references**, joining other **classes** via **references** or **attributes**, and applying filters with ***where*** conditions. The result set is then used during **template** reuse to select one or more options, depending on whether the **input** is single input or a list.

Listing: Query_definition

```
query
  select (distinct)? from class_def_reference =
[Class_def | ReferenceName]

query_join_def += (join class_join_reference =
[Class_def | ReferenceName] on (reference |
attribute)
join_on_left_reference = [Class_reference_def |
Class_attribute_def | ReferenceName]
=
Join_on_right_reference = [Class_reference_def |
Class_attribute_def | ReferenceName])*

(where where_condition_term = Term)?
endquery
```

- *Extras:* **Inputs** may require additional **attributes** that are not defined in the original **metamodel class**. Extras are named elements used to extend a **class** instance within the scope of a specific **input**, without altering the metamodel itself. Extras are limited to primitive types.

Listing: Extra_definition

```
extra id = ID
 type data_type = (number | text | bool …)
 description description_text_term = Term
endextra
```

- *Expected:* **Inputs** may require validation to ensure that certain **attributes** from a **class** are provided when the **template** is reused. The expected element reference a **class attribute**, making it mandatory before the **input** can be propagated further in the **implementation** during **template** reuse.

Listing: Expected_definition

```
expected ref_attribute = [Attriubte_def |
ReferenceName]
 (defaultvalue default_value = Term)?
endextra
```

- *Checking.* Beyond mandatory values, **inputs** may require additional validations to ensure specific constraints are satisfied (e.g., the **input** value must be greater than 0). A checking is a named element that allows the specification of logic-level validations using code blocks, such as for loops, if-else conditions, and mathematical operations. It includes a dedicated **check** statement that evaluates a condition and, if not met, triggers an **else** message to provide feedback during **template** reuse.

Listing: Checking_definition

```
checking id = ID
 checking_content_def +=
(Code_Block |
check condition_term = Term else text_term = Term)*
endchecking
```

Outputs are named elements that represent the **metamodel class** instances (i.e., the resulting models) to be stored into an MDSD tool after reusing a **template**. They can be a single **output** or a list of **outputs** and must be instances of **classes** from the referenced **metamodels**—primitive types are not allowed. Additionally, **outputs** can include **checks** to validate specific conditions and ensure that the generated models satisfy intended constraints. To express the **output** element formally, in the following paragraphs we show the LEMON language syntax using the EBNF, starting with **output** definition and followed by the semantics of the element.

Listing: Output_definition

```
(output | outputlist) id = ID
  type data_type =
  ([Class_reference_def | ReferenceName])
  description description_text_term = Term
  additional_output_def += Checking_def*
(endoutput | endoutputlist)
```

- *Single Output:* Indicates that the **output** contains only one value, which may be **null** or an instance of a **metamodel class**
- *Output list:* Represents an **output** composed of multiple values, either as an empty list or a collection of **metamodel class** instances.
- *Output Value*: **Outputs** must always be instances of **classes** defined in the **metamodel**; primitive types are not permitted
- *Checking*. **Outputs** may include validation logic to ensure they meet specific constraints. These checks reuse the same mechanism defined for **inputs** (see Checking_def Listing).

To transform **inputs** into **outputs**, the LEMON language provides the **apply step**, as a named element that defines the logic for how a specific set of **inputs** should be processed to produce **outputs**. It does so by referencing a reusable **step** definition, which encapsulates the transformation logic. Conceptually, an **apply step** is equivalent to invoking a function in programming languages or calling a transformation rule in MDSD. **Apply step** can be invoked within either a **template implementation** or another **step** definition. **Apply steps** can be configured to iterate over lists, being applied conditionally, or run once depending on the data and logic. Additionally, apply steps map **inputs** and **outputs** from higher-level contexts from/to the referenced **step**. To express the **apply step** element formally, in the following paragraphs we show the LEMON language syntax using the EBNF, starting with **apply step** definition and followed by the definition of step.

Listing: Apply_step_definition

```
applystep step_reference =
  [Step_def | ReferenceName] as id = ID
 (foreach for_each_var_identifier = [ReferenceName]
 in for_each_set_identifier = [ReferenceName])?
 (if apply_step_condition = Term)?
 having as input
  having_as_input_ref += Having_as_input_def*
 (having as output
  having_as_output_ref += Having_as_output_def*)?
endapplystep
```

- *For-each Apply Step*. Applies the referenced **step** multiple times, once for each element in a list, as soon as the required **inputs** are provided.
- *Conditional Apply Step*. Executes the **step** only once, provided that a specified condition evaluates to **true** and the **inputs** are available.
- *Conditional for-each apply step*. Combines both for-each and condition **apply steps**. The **step** is applied to each list element, but only if the condition is satisfied on the level of the individual list element.
- *One-time apply step*. Applies the **step** exactly once, without any looping or conditional logic, as soon as the **inputs** are provided.
- *Having as input*. Specifies the mapping of **input** values from a higher-level context (such as another **step** or the **template implementation**) to the **inputs** of the referenced **step**. These values may reference **inputs**, **outputs**, or use the ***ask*** operator.

Listing: Having_as_input_definition

```
[ Input_def | ReferenceName ] =
(ask | [Input_def | ReferenceName])
```

- *Having as output*. Defines the assignment (from left to right) between the **outputs** of the referenced **step** and those of the enclosing context, such as an **implementation** or **step**. If no explicit **output** mappings are provided, the **outputs** from the referenced **step** are implicitly propagated to the upper-level context.

Listing: Having_as_output_definition

```
[Output_def | ReferenceName ] =
[Output_def | ReferenceName]
```

Finally, a **step** is a named element that encapsulates transformation logic along with its associated **inputs** and **outputs**. It serves as a reusable definition that can be invoked from one or more **apply steps**. Conceptually, a step corresponds to the definition of a transformation rule in MDSD or the definition of a function in programming languages. **Steps** may also include nested **apply steps**, enabling composition and recursion. The core transformation logic is defined within an **apply** block, which is a code block. To express the **step** element formally, in the following listing we show the LEMON language syntax using the EBNF.

Listing X: Step_definition

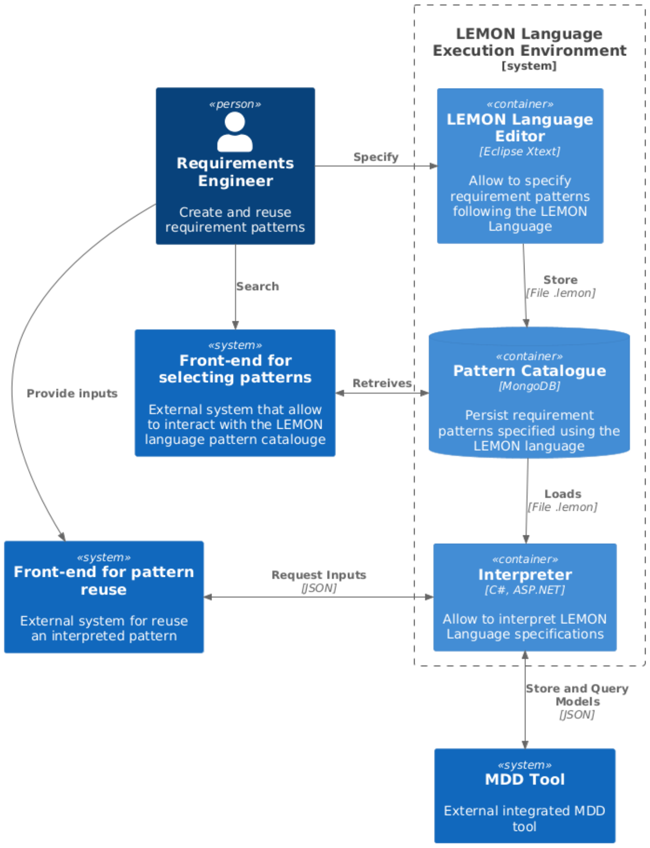
```
step id = ID
 step_inputs += Input_def*
 step_outputs += Output_def*
 step_apply_steps += Apply_step_def*
 apply
```

```
    code_blocks_def = Code_Block
  endapply
endstep
```

We illustrate the **template implementation** in our running example with an implementation for the "List and Search in PDS" **template** (see Annex VII.D)**.** It takes a query name (I_QueryName) and a selected table (I_Table) as **inputs**, and produces a query model (O_QueryModel) as **output**. The **implementation** is composed of two **apply steps**: i) one to create the query model from the given name, and ii) another to relate the created model to the selected table. The **output** of the second **step** is propagated as the final **template implementation output**, having a query model as the result of reusing the "List and Search in PDS" **pattern template.** We provide specific examples on input (see Annex VII.E), output (see Annex VII.F), apply step (see Annex VII.G), and step definitions (see Annex VII.H), to allow the template implementation as required from Whatscount following the LEMON language syntax.

V. Proof of Concept Implementation: Execution Environment for the LEMON Language

To make a proof of concept of the LEMON language, we implemented an execution environment tailored to the language EBNF. This execution environment comprises three main components: the language editor, the pattern catalogue, and the interpreter. We made available the proof of concept of the execution environment in our external repository [21].

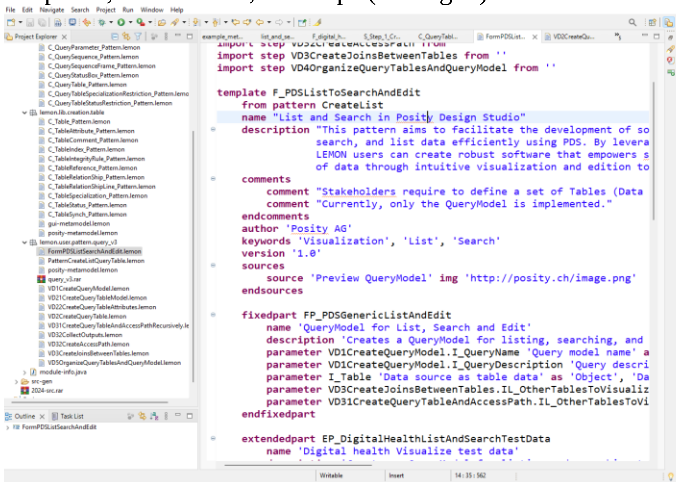


LEMON Language Execution Environment [system]
«person» Requirements Engineer
Create and reuse requirement patterns
Specify
«container» LEMON Language Editor [Eclipse Xtext]
Allow to specify requirement patterns following the LEMON Language
Search
Store [File .lemon]
«system» Front-end for selecting patterns
External system that allow to interact with the LEMON language pattern catalouge
Retreives
«container» Pattern Catalogue [MongoDB]
Persist requirement patterns specified using the LEMON language
Provide inputs
Loads [File .lemon]
«system» Front-end for pattern reuse
External system for reuse an interpreted pattern
Request Inputs [JSON]
«container» Interpreter [C#, ASP.NET]
Allow to interpret LEMON Language specifications
Store and Query Models [JSON]
«system» MDD Tool
External integrated MDD tool

Fig. 1. LEMON Language Execution Environment C4 Container Diagram.

A. *LEMON Language Editor*

The LEMON language editor is built using Eclipse XText [22], relying on the Eclipse Modelling Framework [23]. It provides support for lexical, syntactic, and semantic analysis of LEMON language based on the EBNF grammar introduced earlier. Through this editor, requirements engineers can produce **.lemon** files that capture requirement patterns, templates, metamodels, and steps (see Fig. 2).

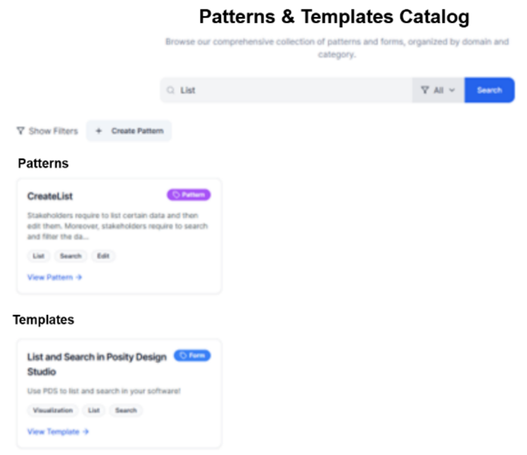


```
import step VD32CreateAccessPath from
import step VD3CreateJoinsBetweenTables from ''
import step VD4OrganizeQueryTablesAndQueryModel from ''

template F_PDSListToSearchAndEdit
    from pattern CreateList
    name "List and Search in Posity Design Studio"
    description "This pattern aims to facilitate the development of so
                search, and list data efficiently using PDS. By levera
                LEMON users can create robust software that empowers s
                of data through intuitive visualization and edition to
    comments
        comment "Stakeholders require to define a set of Tables (Data
        comment "Currently, only the QueryModel is implemented."
    endcomments
    author 'Posity AG'
    keywords 'Visualization', 'List', 'Search'
    version '1.0'
    sources
        source 'Preview QueryModel' img 'http://posity.ch/image.png'
    endsources

    fixedpart FP_PDSGenericListAndEdit
        name 'QueryModel for List, Search and Edit'
        description 'Creates a QueryModel for listing, searching, and
        parameter VD1CreateQueryModel.I_QueryName 'Query model name' a
        parameter VD1CreateQueryModel.I_QueryDescription 'Query descri
        parameter I_Table 'Data source as table data' as 'Object', 'Da
        parameter VD3CreateJoinsBetweenTables.IL_OtherTablesToVisualiz
        parameter VD31CreateQueryTableAndAccessPath.IL_OtherTablesToVi
    endfixedpart

    extendedpart EP_DigitalHealthListAndSearchTestData
        name 'Digital health Visualize test data'
```

Fig. 2. The LEMON Language Eclipse XText Editor.

B. *Pattern Catalogue*

The pattern catalogue is a MongoDB non-relational database that stores the **.lemon** files generated by the language editor. It allows users to search, retrieve, and reuse requirement **patterns** and **templates**. An external front-end interface consumes metadata from stored **patterns**, **templates**, **fixed parts**, **extended parts**, **inputs**, **outputs**, and **steps** enabling recommendation and filtering of relevant requirement specifications (see Fig. 3)



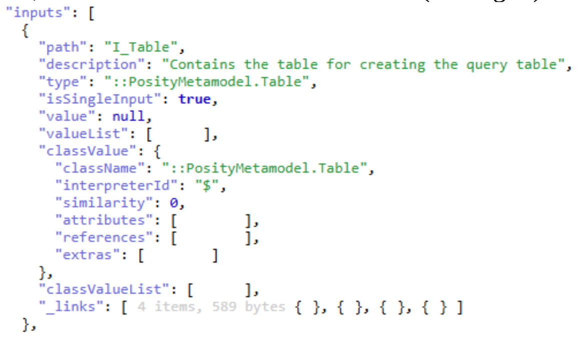
Patterns & Templates Catalog
Browse our comprehensive collection of patterns and forms, organized by domain and category.
List
All
Search
Show Filters
Create Pattern
Patterns
CreateList
Pattern
Stakeholders require to list certain data and then edit them. Moreover, stakeholders require to search and filter the da...
List
Search
Edit
View Pattern →
Templates
List and Search in Posity Design Studio
Form
Use PDS to list and search in your software!
Visualization
List
Search
View Template →

Fig. 3. Mock-up requirement pattern catalogue.

C. *Interpreter*

The LEMON interpreter is a .NET application written in C# that exposes a RESTful API using ASP.NET Core. This API serves both the catalogue (to load and store .lemon files) and external front-ends (to provide **inputs** and drive pattern reuse).

It supports HTTP operations to submit, update, and delete **inputs**, and to retrieve execution feedback (see Fig. 4).



```
"inputs": [
  {
    "path": "I_Table",
    "description": "Contains the table for creating the query table",
    "type": "::PosityMetamodel.Table",
    "isSingleInput": true,
    "value": null,
    "valueList": [      ],
    "classValue": {
      "className": "::PosityMetamodel.Table",
      "interpreterId": "$",
      "similarity": 0,
      "attributes": [        ],
      "references": [        ],
      "extras": [        ]
    },
    "classValueList": [      ],
    "_links": [ 4 items, 589 bytes { }, { }, { }, { } ]
  },
```

Fig. 4. LEMON Interpreter RESTFul API Example.

The interpreter supports both basic and advanced execution operations. Core capabilities include arithmetic operations, memory management, and control flows such as conditionals and loops. More advanced features include undo/redo handling, automatic data bindings among **inputs** and **outputs,** and runtime error detection. Once a **pattern template** is loaded, the interpreter initializes a context-bound **template** instance, generates corresponding **input**/**output** objects, and evaluates **steps**, **queries**, and **checks** (see Fig. 5).

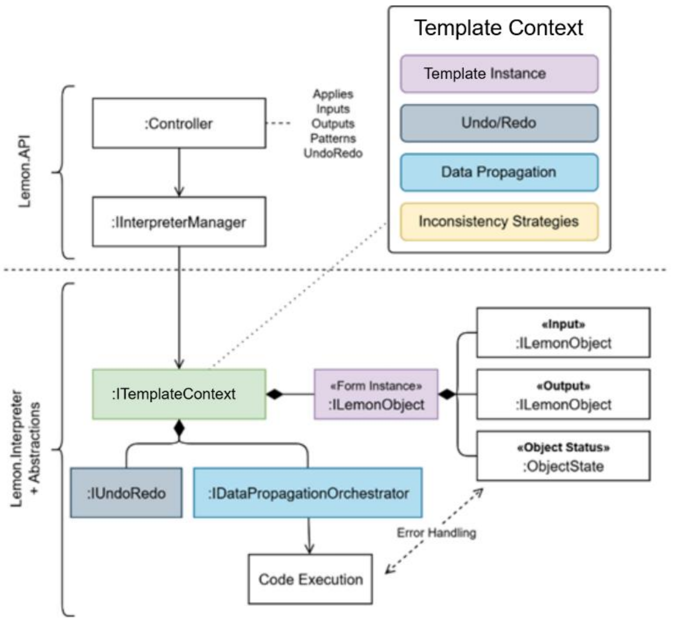


Fig. 5. LEMON Interpreter Abstractions.

The code execution workflow is as follows: i) the interpreter identifies **inputs** marked as "***ask***"; ii) as inputs are provided, values propagate to dependent **apply steps** via having as input; iii) nested **apply steps** recursively evaluate available **inputs**; iv) once **inputs** are resolved, **apply** are executed; v) finally, **outputs** are validated and stored. This process is triggered whenever an **input** value changes (see Fig. 6).

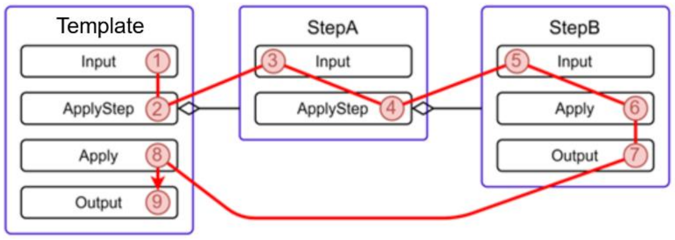


Fig. 6. LEMON Code Execution.

As **inputs** and **outputs** are linked to the MDSD tool's **metamodel**, the interpreter can query data via **queries** and persist generated **outputs** into the tool's environment, completing the requirement pattern reuse for MDSD tools. In Fig. 7, we show how the query model output after reusing the **implemented template** of our running example at Whatscount is stored in PDS.

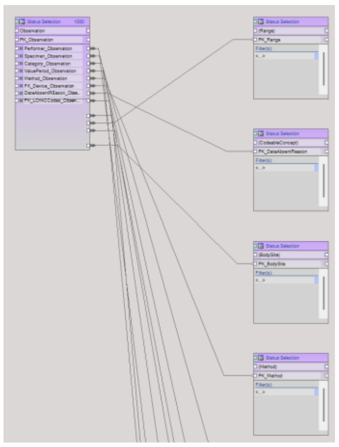


Fig. 7. Example Model generated into an integrated MDSD tool: A Query Model in PDS.

VI. Conclusions and Future Work

In this paper, we introduced the LEMON language—a domain-specific language designed to bridge the gap between reusable requirement patterns and Model-Driven Software Development (MDSD) tools. We presented the EBNF grammar of the language, described how it enables the specification of **patterns** and **templates**, and showed how it connects to MDSD tool **metamodels** through **template implementations**. Furthermore, we demonstrated how the LEMON language can be executed to allow requirement patterns reuse and integration with MDSD tools by proposing an LEMON execution environment. The LEMON execution environment includes a language editor, catalogue, and interpreter.

As part of our ongoing work on the LEMON language, we identify open research directions as future work:

RQ1. How do requirements engineers use the LEMON language to specify requirement patterns for MDSD tools? While this paper provides the technical foundation and language design, questions remain regarding the actual usability of LEMON in real-world settings. Empirical studies are needed to observe how requirements engineers create, modify, and reuse patterns using LEMON, and to evaluate their efficiency and effectiveness when doing so. Additionally, aspects such as support for pattern composition and integration with various MDSD tools must be explored to understand the practical impact of LEMON in diverse MDSD tools.

RQ2. Can the specification of requirement patterns be automated using the LEMON language as a foundation? A key advantage of having a domain-specific language is the possibility of automating the generation and validation of requirement patterns. Inspired by recent work using Large Language Models (LLMs) for requirement patterns specification and reuse [9], this research question aims to investigate whether Artificial Intelligence techniques such as LLMs can be leveraged to automatically generate LEMON-conformant specifications. Such specifications could then be checked for consistency using the language's EBNF grammar. This research question would explore how the LEMON language can be used to systematically build catalogs of patterns, supported by exploratory research on existing application templates, requirement samples, and MDSD tool (a.k.a. LowCode/NoCode tools) resources available in industry [13], [14].

RQ3. How do requirements engineers interact with the pattern catalogue and reuse interface? The LEMON language serves as an enabler for specifying, retrieving, and reusing requirement patterns within MDSD tools. However, the usability and effectiveness of the external front-ends for interacting with the catalogue and for reusing the requirement patterns specified in LEMON remain open questions. Future research should include empirical evaluations and user studies to assess how well requirements engineers interact with these components. Design iterations and interface prototypes will also be necessary to determine how these external front-end systems can best support the reuse of requirement patterns in MDSD tools.

These research questions aim to extend the technical contributions of this paper and establish a pathway toward realizing the full potential of the LEMON language in bridging the gap between requirements engineering and model-driven software development through requirement pattern reuse.

VII. Annex: LEMON Language Example

This annex presents the listings for illustrating the use of the LEMON language at Whatscount. Each listing is referenced in the text.

A. Pattern Specification Example

```
pattern CreateList
 metadata
   name "Searchable and Editable List"
  description "This pattern supports presenting
structured data in a list that users can search,
filter, and edit—common in scenarios requiring data
overview and manipulation, like user management or
inventory control."
  goal "Enable users to efficiently view, search
and edit structured data records."
  author "Whatscount"
  keywords "Data List", "Search functionality",
"Editable Entries", "Filtering"

  sources
      source "Internal project experience with data
management" url "https://..."
      source "UI/UX reference diagram for list
manipulation" img "example.png"
  endsources

  classified as "Functional Requirement"

 endmetadata
endpattern
```

B. Template Specification Example

```
template F_PDSListToSearchAndEdit
  from pattern CreateList
  name "List and Search in PDS"
  description "This template implements the
Searchable and Editable List pattern within the
Posity Design Studio (PDS). It provides requirement
engineers with a structured approach to define
QueryModels that enable dynamic listing, searching,
and editing of domain-specific data. By guiding
users through the setup of core data tables and
their relationships, this template supports the
efficient visualisation of structured records"
 author "Whatscount and Posity AG"
 keywords "Visualization", "QueryModel", "Search",
"PDS"
 version "1.0"

 fixedpart FP_PDSGenericListAndEdit
   name "Generic QueryModel for Listing and
Editing"
    description "This fixed part defines the
essential structure for creating a QueryModel in
PDS for listing, searching, and editing data"
    parameter I_QueryName "Name of the query model"
as "Data", "Form", "Object"
    parameter I_Table "Primary data source" as
"Object", "Data"
 endfixedpart

 extendedpart EP_DigitalHealthListAndSearchTestData
   name "Digital Health Test Data View"
   description "This extended part specialises the
generic list and search pattern template for
digital health applications. As a result, a
QueryModel is created focused on visualizing and
navigating test data, such as vital signs or lab
results."
   parameter I_QueryName "What is the name of the
health test to visualize?" as "Health Test", "Blood
Pressure Test", "Cell count panel"
   parameter I_Table "Where is the test data
stored?" as "Blood pressure observation", "Vital
sign observation", "EPD"
 endextendedpart

 implementation …
 endimplementation
endtemplate
```

C. Metamodel Definition Example

```
metamodel PosityDesignStudioMetamodel
  class Table
   attribute Name
      type text
      default "Default table name"
      primarykey true
   endattribute
 endclass

 class QueryModel
```

```
   attribute Name
       type text
       default "Default Query model name"
       primarykey true
   endattribute
   reference FKTableOfQueryModel
       type Table
       ordered false
       unique false
       keys Table.Name
       lowerbound 0
       upperbound *
    endreference
  endclass
endmetamodel
```

D. Template Implementation Example

```
template F_PDSListToSearchAndEdit from pattern
CreateList
  …
implementation  PDS_Implementation
 tools PosityDesignStudioMetamodel

 input I_QueryName
  type text
  description "Name of the query model"
  value ask
 endinput

 input I_Table
   type PosityDesignStudioMetamodel.Table
   … // See Input Definition example
 endinput

 output O_QueryModel
   type PosityDesignStudioMetamodel.O_QueryModel
   … // See Output Definition example
 endoutput

 //First the query model need to be created based
 //On the provided name
 applystep S_CreateNamedQueryModel
           as s_first_create_query_model
   having as input
      I_Name = I_QueryName
 endapplystep

 //Then: The created query model is related to
 //the selected Table (see example Step def)
 applystep S_RelateQueryModelWithTable
          as s_second_relate_table_to_query_model
 having as input
  I_SelectedTable = I_Table
  I_QueryModel = s_first_create_query_model.O_QueryModel
 having as output
  O_QueryModel = O_RelatedQueryModel
 endapplystep

endimplementation
endtemplate
```

E. Input definition example

```
input I_Table
  type PosityDesignStudioMetamodel.Table
  description "Contains a Table for creating a
QueryModel out of it"
  value ask
  query
    select distinct from PosityDesignStudioMetamodel.Table
  endquery
  expected I_Table.Name endexpected
  extra e_is_header
    type bool
    description "Determines wheter the selected
Table is the Header of the QueryModel"
  endextra
  checking C_has_a_name
    check I_Table.Name <> "" else "The table name
can not be an empty string"
  endchecking
endinput
```

F. Output definition example

```
output O_QueryModel
  type PosityDesignStudioMetamodel.QueryModel
  description "Final Query Model"
  checking C_has_valid_name
   check O_QueryModel <> null else "QueryModel Name
can not be empty"
  endchecking
endoutput
```

G. Apply step definition example

```
applystep S_RelateQueryModelWithTable
          as s_second_relate_table_to_query_model
 having as input
  I_SelectedTable = I_Table
  I_QueryModel = s_first_create_query_model.O_QueryModel
 having as output
  O_QueryModel = O_RelatedQueryModel
endapplystep
```

H. Step definition example

Listing X: Example Step_definition

```
step S_RelateQueryModelWithTable
 input I_SelectedTable
   type PosityDesignStudioMetamodel.Table
   …
 endinput
 input I_QueryModelToRelate
   type PosityDesignStudioMetamodel.QueryModel
   …
 endinput
 output O_RelatedQueryModel
   type PosityDesignStudioMetamodel.QueryModel
   …
 endoutput
 apply
   O_RelatedQueryModel = I_QueryModelToRelate
   relate O_RelatedQueryModel
   to I_SelectedTable in FKTableOfQueryModel
 endapply
endstep
```

ACKNOWLEDGMENT

We would like to thank all the people who contributed valuable feedback during the design of the LEMON language. In particular, we express our gratitude to Tobias Ulrich, Veronika Kamber, Olivia Schöb, Patricia Fiechter, and Eslem Küçükkaya for their work on the LEMON language user manual, which is provided as a supplementary resource in the external repository linked to this paper. This research was fully supported by the ZHAW Institute of Computer Sciences (InIT), School of Engineering, and the Innosuisse Flagship SHIFT project.

REFERENCES

[1] A. Gupta, G. Poels, and P. Bera, "Using Conceptual Models in Agile Software Development: A Possible Solution to Requirements Engineering Challenges in Agile Projects," *IEEE Access*, vol. 10, pp. 119745–119766, 2022, doi: 10.1109/ACCESS.2022.3221428.

[2] T. N. Kudo, R. F. Bulcão-Neto, and A. M. R. Vincenzi, "Requirement patterns: a tertiary study and a research agenda," *IET Software*, vol. 14, no. 1, pp. 18–26, Feb. 2020, doi: 10.1049/iet-sen.2019.0016.

[3] P. Mahendra and A. Ghazarian, "Patterns in the Requirements Engineering: A Survey and Analysis Study," in *WSEAS Transactions of Information Science and Applications*, 2014, pp. 214–230.

[4] S. Renault, O. Mendez-Bonilla, X. Franch, and C. Quer, "PABRE: Pattern-based Requirements Elicitation," in *2009 Third International Conference on Research Challenges in Information Science*, IEEE, Apr. 2009, pp. 81–92. doi: 10.1109/RCIS.2009.5089271.

[5] K. Kumar and Ra. K. Saravanaguru, "CONTEXT AWARE REQUIREMENT PATTERNS (CaRePa) METHODOLOGY AND ITS EVALUATION," *Far East Journal of Electronics and Communications*, vol. 16, no. 1, pp. 101–117, Feb. 2016, doi: 10.17654/EC016010101.

[6] L. Sardi, A. Idri, L. Redman, H. Alami, and J. Fernández-Alemán, "A Reusable Catalog of Requirements for Gamified Mobile Health Applications," in *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering*, SCITEPRESS - Science and Technology Publications, 2022, pp. 435–442. doi: 10.5220/0011071700003176.

[7] A. Sleimi, M. Ceci, M. Sabetzadeh, L. C. Briand, and J. Dann, "Automated Recommendation of Templates for Legal Requirements," in *Proceedings of the IEEE International Conference on Requirements Engineering*, IEEE Computer Society, Aug. 2020, pp. 158–168. doi: 10.1109/RE48521.2020.00027.

[8] C. Denger, D. M. Berry, and E. Kamsties, "Higher quality requirements specifications through natural language patterns," in *Proceedings 2003 Symposium on Security and Privacy*, IEEE, 2003, pp. 80–90. doi: 10.1109/SWSTE.2003.1245428.

[9] X. Franch, S. Gnesi, F. Paccosi, C. Quer, and L. Semini, "Leveraging Requirements Elicitation through Software Requirement Patterns and LLMs," in REFSQ2025, 2025, pp. 261–276. doi: 10.1007/978-3-031-88531-0_19.

[10] I. Darif, G. El Boussaidi, S. Kpodjedo, and A. Paz, "UTL: A Unified Language for Requirements Templates," in *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*, 2025, pp. 1489–1506. doi: 10.1145/3672608.3707911.

[11] S. Robertson, "Requirements Patterns Via Events/Use Cases," in PLoP, 1996, pp. 1-16.

[12] A. R. da Silva *et al.*, "A pattern language for use cases specification," in *Proceedings of the 20th European Conference on Pattern Languages of Programs*, New York, NY, USA: ACM, Jul. 2015, pp. 1–18. doi: 10.1145/2855321.2855330.

[13] Mendix, "Mendix Sample and Starter App Catalogue." Accessed: Mar. 30, 2025. [Online]. Available: https://marketplace.mendix.com/link/contenttype/102,109

[14] OutSystems, "OutSystems: Application Templates." Accessed: Jul. 11, 2024. [Online]. Available: https://success.outsystems.com/documentation/11/building_apps/application_templates/

[15] D. Mosquera, O. Pastor, and J. Spielberger, "LEMON: A Tool for Enhancing Software Requirements Communication through Requirements Pattern-based Modelling Assistance," in *Posters & Tools Track at REFSQ2024*, 2024.

[16] I. Darif, G. El Boussaidi, and S. Kpodjedo, "On the Automated Generation of UI for Template-based Requirements Specification," in *MO2RE*, 2025.

[17] S. Sendall and W. Kozaczynski, "Model transformation: the heart and soul of model-driven software development," *IEEE Softw*, vol. 20, no. 5, pp. 42–45, Sep. 2003, doi: 10.1109/MS.2003.1231150.

[18] Posity AG, "Posity Design Studio Homepage." Accessed: May 31, 2025. [Online]. Available: https://posity.ch

[19] Wikipedia, "Extended Backus-Naur Form." Accessed: May 31, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Extended_Backus–Naur_form

[20] Eclipse Foundation, "Ecore: A Metamodel for Models." Accessed: Mar. 07, 2024. [Online]. Available: https://wiki.eclipse.org/Ecore

[21] D. Mosquera, M. Ruiz, and A. Martakos, "A Domain-Specific Language For Specifying Requirement Patterns for Model-Driven Development - EBNF and Execution Envrionment POC." Accessed: Jun. 01, 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15601580

[22] XText, "XText Eclipse Homepage." Accessed: May 31, 2025. [Online]. Available: https://eclipse.dev/Xtext

[23] Eclipse, "Eclipse Modelling Framework." Accessed: May 31, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Eclipse_Modeling_Framework