Leveraging LLMs to support co-evolution between definitions and instances of textual DSLs

Weixing Zhang[1,*,†], Regina Hebig[2,*,†] and Daniel Strüber[1,3,*,†]

[1]Chalmers University of Technology and University of Gothenburg, Hörselgången 5, 417 56 Göteborg, Sweden
[2]Universität Rostock, Albert-Einstein-Straße 22, 18059 Rostock, Germany
[3]Radboud University, Toernooiveld 212, 6525 EC Nijmegen, The Netherlands

Abstract

Software languages evolve over time for various reasons, such as the addition of new features. When the language's grammar definition evolves, textual instances that originally conformed to the grammar become outdated. For DSLs in a model-driven engineering context, there exists a plethora of techniques to co-evolve models with the evolving metamodel. However, these techniques are not geared to support DSLs with a textual syntax — applying them to textual language definitions and instances may lead to the loss of information from the original instances, such as comments and layout information, which are valuable for software comprehension and maintenance. This study explores the potential of Large Language Model (LLM)-based solutions in achieving grammar and instance co-evolution, with attention to their ability to preserve auxiliary information when directly processing textual instances. By applying two advanced language models, Claude-3.5 and GPT-4o, and conducting experiments across seven case languages, we evaluated the feasibility and limitations of this approach. Our results indicate a good ability of the considered LLMs for migrating textual instances in small-scale cases with limited instance size, which are representative of a subset of cases encountered in practice. In addition, we observe significant challenges with the scalability of LLM-based solutions to larger instances, leading to insights that are useful for informing future research.

Keywords

Co-Evolution, textual DSLs, LLM

1. Introduction

Domain-specific languages (DSLs) are useful tools to describe and solve problems in a specific application domain. As domain knowledge evolves and requirements change, DSLs often need to evolve accordingly [1]. For example, features may be added and existing functionality may be adjusted, leading to a need to update the definition of the DSL, to introduce new language constructs and modify existing ones. When the definition of a DSL evolves, existing instances face challenges: they may contain constructs that no longer conform to the new definition and require appropriate modification, or they may need to additions to support newly introduced required language elements [2, 3, 4]. While the model-driven engineering community has developed numerous approaches for metamodel-instance co-evolution [5], these works are generally focused on metamodel-based language definitions, usually in the context of graphical DSLs [5]. In practice, there is an ongoing trend towards textual DSLs, which can emulate the *look and feel* of familiar general-purpose languages and are easy to integrate with standard developer tools for versioning, differencing, and merging. These textual DSLs, developed in frameworks like Xtext, Langium, and textX, are technically defined through grammars and instantiated as textual instances.

Dedicated approaches to co-evolving textual instances are scarce. One possible way to address co-evolution of textual instances is by using the available metamodel-based approaches. To that end, the original instance needs to be parsed into the form of a model and transformed back into textual form after the model is co-evolved. However, this approach leads to information loss: during the transformation

First Large Language Models for Software Engineering Workshop (LLM4SE 2025), Koblenz, Germany
weixing.zhang@gu.se (W. Zhang); regina.hebig@uni-rostock.de (R. Hebig); danstru@chalmers.se (D. Strüber)
https://wilson008.github.io/ (W. Zhang);
https://se.informatik.uni-rostock.de/team/lehrstuhlinhaber/prof-dr-rer-nat-regina-hebig/ (R. Hebig);
https://www.danielstrueber.de/ (D. Strüber)
0000-0003-2890-6034 (W. Zhang); 0000-0002-1459-2081 (R. Hebig); 0000-0002-5969-3521 (D. Strüber)

process between textual instance and model, auxiliary information in the original instances, such as code comments and formatting styles, cannot be retained [6][7]. While this information does not affect program functionality, it serves a critical purpose during tasks such as code maintenance, debugging, and understanding design intent [8]. Hence, there is arguably a need to preserve such information during the co-evolution of instances.

In recent years, Large Language Models (LLMs) have demonstrated exceptional capabilities in code understanding, transformation, and generation [9] [10]. These models not only perform well at tasks that require understanding of code structure, but also capture contextual information like comments. As such, they seem a particularly well-suited for addressing the co-evolution problem for textual languages.

In this paper, we investigate the use of LLMs to support the co-evolution of grammar definitions and instances for textual DSLs. We focus on grammar definitions and instances developed using Xtext [11], a framework that is rooted in the Eclipse ecosystem and is particularly widely used in the MDE community. We harness our recently published dataset on Xtext-based language evolution cases retrieved from GitHub [12], which allowed us to identify a collection of critical real-life cases in which automated support for co-evolution would be desirable. We address and answer the following the following research questions:

RQ1: How can we use LLMs to automate the co-evolution of instances with evolving Xtext grammars?

To address this RQ, we explored an approach that uses LLMs to analyze differences between original and evolved grammars, and generate evolved instances that conform to the new grammar while preserving auxiliary information. We implemented this approach using GPT-4o and Claude-3.5, with a dedicated prompt and automated workflow.

We evaluate the two LLM-based solutions on seven case languages, focusing on the following research questions:

RQ2: How does instance size affect the capability of LLMs to produce correct solutions when co-evolving textual instances?

RQ3: How does instance size affect LLMs' capability in preserving auxiliary information during DSL co-evolution?

The contribution of this paper is an exploration of the potential of LLM for the co-evolution of grammar and instances of DSLs, including auxiliary information in instances such as comments. We evaluated capability differences between two mainstream LLMs in DSL co-evolution tasks and analyzed the advantages and limitations of purely LLM-based DSL co-evolution methods, providing direction for future research.

2. Problem Description

Xtext is an Eclipse-based framework for developing languages[1]. In Xtext projects, there is a close relationship between grammar and metamodel: when the grammar evolves, a corresponding new metamodel can be automatically generated from the new grammar, and vice versa. In our previous work, we showed how the customized adaptations in the original grammar can be preserved [13][14] to complete the grammar co-evolution. However, after the grammar evolves, the instances that originally conformed to it may no longer conform to it. It would be possible to use existing techniques to co-evolve models with evolving metamodels. For that, the textual instance can be parsed using the original grammar to gain the model representation (i.e., a .xmi file) that conforms to the original metamodel. Then existing model migration techniques (such as EMFMigrate [15]) can be used to transform this model into a model that conforms to the new metamodel. Finally, we can transform the migrated model back into a textual instance that conforms to the new grammar.

However, in this process, the auxiliary information (e.g., comments and code formatting) in the original instance is discarded during the transformations. As an example, there is a "15 minutes tutorial"[2] on the official Xtext website which provides an example called `Domainmodel`, which includes two

[1] https://eclipse.dev/Xtext/index.html

[2] https://eclipse.dev/Xtext/documentation/102_domainmodelwalkthrough.html

versions of grammar and their corresponding instances, where the second version adds five grammar rules. Consider a slightly modified version of the instance from the tutorial example with auxiliary information in different places in the instance, shown in Listing 1. Line 10 is empty, which, in the case of entities with many more attributes, is a useful way to group them. The definition of instance `HasAuthor` has been compressed from originally four lines to a single line (line 14) by removing whitespace, making the overall instance more compact and easier to overview. Line 19 uses comments as a way to discard a part from the instance that might potentially be included again at a later time (outcommenting). Line 24 contains an additional comment in a style commonly used to add rationale and context to individual statements. More subtlely, lines 9 and 11 use a different type of indentation than the rest of the instance, based on tabs, which could be the result of an ongoing manual review and refactoring.

Listing 1: Instance conforming to grammar before evolution.

```
/**
 * This is the example before the evolution.
 * This is the header.
 * */
datatype String
/* this is the first comment, added by me */
entity Blog {
  title: String
  many posts: Post

}
entity HasAuthor { author: String }
entity Post extends HasAuthor {
  title: String
  content: String
  //many comment: Comment
  many comments: Comment
}
entity Comment extends HasAuthor {
  content: String // this is the second comment
}
```

Listing 2: Co-evolved instance following grammar after evolution. Layout and comments are lost.

```
datatype String;

entity Blog {
  title: String,
  many posts: Post
}

entity HasAuthor {
  author: String
}

entity Post extends HasAuthor {
  title: String,
  content: String,
  many comments: Comment
}

entity Comment extends HasAuthor {
  content: String
}
```

Traditional co-evolution approaches in the MDE sphere focus on co-evolution on the abstract syntax level, i.e., the impact of new and changed meta-model classes and relationships to model elements. Concrete syntax information without an abstract syntax counterpart – that is, auxiliary information such as comments and whitespace – is not covered, and hence lost in the process. This is illustrated by Listing 2 showing the result of applying such an approach to the exam instance, which leads to the loss of all comments and formatting information.

3. Related Work

DSL co-evolution. Hebig et al. [5] present a survey of approaches to co-evolve models with evolving metamodels, summarizing a multitude of approaches ranging from languages specialized to the automated generation of model transformations for dealing with non-breaking changes [16], to approaches that allow the definition of migration strategies, such as EMFMigrate[15] and Epsilon Flock [17]. Tolvanen et al. [18] proposed a framework for evaluating tool support for DSL co-evolution. but primarily focuses on graphical DSLs and evaluates tool-level support capabilities. In contrast, our work focuses on textual DSLs, exploring the potential of LLMs as a novel technical approach for textual DSL evolution challenges, particularly emphasizing the preservation of auxiliary information such as comments and layout during co-evolution.

Application of LLM in MDE. Di Rocco et al. [19] conducted a systematic review of LLM applications in Model-Driven Engineering (MDE). They analyzed the current state of LLM applications in tasks such as model completion, generation, and evolution and proposed a technical framework to guide

Table 1
Case languages and their basic information.

Name	Domain	Category
xtext-orm[1]	Database	Data Management and Databases
xtext-dnn[2]	Deep learning	Artificial Intelligence and Machine Learning
smart-dsl[3]	Blockchain	Security and Networking
mongoBeans[4]	Database access	Data Management and Databases
elite-se.xtext.languages.plantuml[5]	GPL	Programming Languages
isis-script[6]	Java ISIS applications	Software Development and Engineering
CheckerDSL[7]	Test cases	Testing and Verification

[1] https://github.com/blue995/xtext-orm.
[2] https://github.com/Xpitfire/xtext-dnn.
[3] https://github.com/bujosa/smart-dsl.
[4] https://github.com/szarnekow/mongoBeans.
[5] https://github.com/elite-se/elite-se.xtext.languages.plantuml.
[6] https://github.com/vaulttec/isis-script.
[7] https://github.com/ryanignatius/CheckerDSL.

LLM adoption in MDE. Although their research focuses on model-level evolution, their methodological framework, particularly the insights into prompt engineering design, provides a valuable reference for our handling of textual instance evolution. Kebaili et al. [20] explored the use of Large Language Models to address the co-evolution of code impacted by metamodel evolution. They proposed a prompt engineering-based approach which achieved an accuracy of 88.7%, reaching 95.2% for complex change scenarios across seven Eclipse projects. While their work focuses on co-evolution between metamodels and generated code, our study addresses co-evolution between grammar definitions and textual instances. Although the research objectives differ, their results confirm the potential of LLMs in handling software artifact evolution problems, which aligns with our findings.

4. Methodology

The research methodology consists of three major steps: First, we selected seven diverse DSLs as evaluation objects. We then designed an LLM-based method to co-evolve grammar and instances and implemented it as two solutions based on two LLMs (i.e., Claude-3.5 and GPT-4o). Third, we applied these two solutions to the selected case languages and analyze the results to evaluate their co-evolution potential. The following subsections detail each step.

4.1. Case Language Selection

We searched for case languages from an available dataset [12], since this dataset is specifically dedicated to Xtext-based DSLs. We limited our selection to repositories that contain both Xtext files and instance files. From the commit records of the repository, we can see that the Xtext files and instance files contained may have many commits. For example, in the language *elite-se.xtext.languages.plantuml*, the grammar file *PlantUML.xtext* has 63 commits. For each language, we decided to pick a grammar from the commit that is closest to the present and ensure that there must be an instance that complies with the grammar, otherwise, we will look for the grammar in earlier commits. The found grammar is the evolved grammar. Then, we continue to look for a grammar that differs from this version in earlier commits, and which has instances that comply with it. We identified seven case languages from the dataset. Their basic information is shown in Table 1, and the grammars before/after evolution and the instances that comply with this grammar found in their repositories are shown in Table 2.

In Table 2, *Grammar 1* is a grammar with an earlier commit time, which is regarded as the grammar before the evolution, while *Grammar 2* is a grammar with a later commit time, which is referred to as the evolved grammar. Similarly, *Instance 1* is an instance with an earlier commit time, which is the object of LLM evolution operation (we call it the ***original instance***), while *Instance 2* is an instance with a later commit time, which conforms to the evolved grammar, but may not be an evolved version of *Instance 1*, because the author of the instance may add or delete content irrelevant to the evolution. *Instance 2* serves only as a reference. We will discuss this situation in the discussion section.

Table 2
Selected grammar and instance versions of the case languages. "Grammar 1" denotes the grammar version before the evolution and "Grammar 2" denotes grammar version after the evolution. The same rule applies to the number after "Instance".

	Grammar 1		Instance 1			Grammar 2		Instance 2		
Name	Date[1]	ID[2]	Date[1]	ID[2]	Lines[3]	Date[1]	ID[2]	Date[1]	ID[2]	Lines[3]
xtext-orm	2018-06-21	f84e2b3	2018-06-21	181dcd7	72	2018-06-23	7cb74b2	2018-06-23	7cb74b2	85
xtext-dnn	2016-12-13	a4912d2	2016-12-13	a4912d2	33	2016-12-17	15ccbc0	2016-12-17	15ccbc0	42
smart-dsl	2023-07-22	64259ba	2023-07-22	64259ba	30	2023-07-31	23424ac	2023-07-31	23424ac	30
mongoBeans	2012-06-04	279ecc8	2012-06-04	279ecc8	52	2012-06-06	9f8360b	2012-06-06	9f8360b	29
plantuml[1]	2020-07-12	98f445d	2020-07-12	98f445d	60	2020-07-13	a41d99f	2020-07-13	a41d99f	60
isis-script	2015-07-18	2546850	2015-07-22	1c88416	98	2015-09-05	f0380e8	2015-09-05	f0380e8	68
CheckerDSL	2015-05-03	55911bf	2015-05-03	55911bf	173	2015-07-27	3fa6e6d	2015-07-27	3fa6e6d	181

[1] To shorten the table, we abbreviate the language name "elite-se.xtext.languages.plantuml" to "plantuml".
[2] "Date" = the commit date of the grammar file.
[3] "ID" = the commit ID of the grammar file's commit.

4.2. Solution Design

To implement and evaluate this approach, we selected two mainstream LLMs: Claude-3.5 and GPT-4o, given their demonstrated capabilities in code understanding and generation tasks. We developed Python-based automation scripts that interact with these models through their respective APIs, using specially designed prompts to guide the LLMs in analyzing grammar differences and performing instance evolution. The Python-based scripts can be found in the supplemental materials at https://osf.io/dhjw2/. The approach takes three inputs: the original grammar, an instance conforming to it (i.e., the instance to be evolved), and the evolved grammar. The scripts handle input file processing, manage communication with the LLMs, and save the generated evolved instances. The following section details our prompting strategy, which is critical to achieving successful co-evolution.

Prompt Optimization. To obtain a prompt that can effectively guide the LLM to co-evolve an instance, we started with an initial prompt that we iteratively refined in the two LLM solutions based on the example *Domainmodel* in the official "15 minutes tutorial" of Xtext. In this example, we made some changes to the instance before evolution and the grammar after evolution. The changes made to the instance before evolution have been introduced in Section 2, and we will introduce the changes to the grammar after evolution below. In each iteration, we used the prompt to drive the LLM to evolve the instance and then observe whether there are problems with the output instance, e.g., incorrectly modified elements. If there were problems with the output instance, we adjusted and optimized the prompt according to the problem and entered the next iteration.

LLMs are generally affected by non-determinism, which we need to account for when evaluating the capability of the resulting approach. To this end, we repeated the co-evolution. When the instance was correctly co-evolved, we performed nine more co-evolution runs with the same prompt. Considering the uncertainty of LLM outputs, we decided that when at least six of the ten runs output good instances, we would use this version of the prompt as the final version. An instance is considered good if it follows the evolved grammar and retains auxiliary information. The grammar before evolution is shown in List 3, which contains five grammar rules.

The same tutorial provides an evolved grammar. This evolution is adding five new grammar rules, but does not involve any changes to the symbols in the grammar. To make the evolution changes also reflected in the symbol changes, we make two modifications to the evolved version, i.e., 1) in the `Entity` rule, we add commas (',') to the attribute features to separate it, instead of just spaces; 2) add a semicolon (';') as a terminator at the end of the `DataType` rule. In addition, we also deliberately added an optional attribute called `default` in the grammar rule `Feature` which means that LLMs need to identify more changes in the grammar. The final evolved grammar is shown in Listing 4.

The tutorial also provides an instance that conforms to the grammar before the evolution. But as we

Listing 3: The grammar of Domainmodel before the evolution.

```
...

Domainmodel:
    (elements+=Type)*;

Type:
    DataType | Entity;

DataType:
    'datatype' name=ID;

Entity:
    'entity' name=ID ('extends' superType=[Entity])?
        '{'
        (features+=Feature)*
    '}';

Feature:
    (many?='many')? name=ID ':' type=[Type];
```

Listing 4: The grammar of Domainmodel after the evolution.

```
...
Domainmodel: (elements+=AbstractElement)*;
PackageDeclaration:
    'package' name=QualifiedName '{'
        (elements+=AbstractElement)* '}';
AbstractElement:
    PackageDeclaration | Type | Import;
QualifiedName: ID ('.' ID)*;
Import:
    'import' importedNamespace=
        QualifiedNameWithWildcard;
QualifiedNameWithWildcard: QualifiedName '.*'?;
Type: DataType | Entity;
DataType: 'datatype' name=ID ';';
Entity:
    'entity' name=ID ('extends' superType=[Entity|
        QualifiedName])? '{'
        (features+=Feature  (',' features+=Feature)*)
            ?'}';
Feature:
    (many?='many')? name=ID ':' type=[Type|
        QualifiedName] ('(' default=ID ')')?;
```

mentioned in Section 2, we added comments and format information to the instance before evolution (as shown in Listing 1) to evaluate whether the solution can preserve auxiliary information during co-evolution. We added four comments, one of which is changed from a normal instance line. In addition, we added an empty line and two tabs at different locations and put a multi-line code block into one line. Under the guidance of the prompting text, LLMs are expected to correctly identify this formatting information and replay it in the evolved instance.

4.3. Evaluation

In Step 3, we executed the solution and evaluated the results. We applied the optimized Python script and prompt text to seven case languages, using two models, Claude-3.5 and GPT-4o, to generate two evolution instances for each language. In order to comprehensively evaluate the quality of the evolution instances, we established a multi-dimensional evaluation index system covering three key aspects: grammar correctness, evolution accuracy, and auxiliary information retention ability:

Grammar Correctness Metrics: We consider one metric, *#LineErr*: The number of lines containing grammar errors in the evolved instance. This metric measures the conformity of the instance generated by LLM with the evolved grammar (Grammar 2). A value of "0" indicates full conformity with the new grammar, and a larger value indicates a lower degree of grammar conformity.

Evolution Accuracy Metrics: We consider two metrics: (i.) *#LineEvl*: Count of lines of instance 1 that required change and are correctly evolved. This metric reflects the ability of LLM to correctly identify and process the grammar elements that need to be changed. (ii.) *#LineEvlWrg*: Count of lines of instance 1 that are lost (or incorrectly evolved). This metric measures the number of rows that were incorrectly modified, missed necessary modifications, or introduced unnecessary modifications during the evolution process.

Auxiliary Information Preservation Metrics: We consider four metrics: (i.) *#LineCmtLost*: Count of lines of instance 1 with comments that are lost. (ii.) *#LineCmtSave*: Count of lines of instance 1 with comments that are maintained. (iii.) *#LineFmtLost*: Count of lines of instance 1 with format information that is lost. (iv.) *#LineFmtSave*: Count of lines of instance 1 with format information that is maintained.

All Python scripts, prompts, grammars, original instances, and LLM-generated instances are available in our supporting materials https://osf.io/dhjw2/.

Table 3
This table shows the results of the Claude-based solution in seven case languages. For each case language, we compare the instance output by Claude-3.5 with *instance 1* and then count the various metrics of the evolution. We execute the evolution ten runs and then average the ten counts.

Name	#LineErr	#LineEvl	#LineEvlWrg	#LineCmtLost	#LineCmtSave	#LineFmtLost	#LineFmtSave
xtext-orm	0	11	0	0	1	0	72
xtext-dnn	0	15	0.1	0	0	0.1	32.9
smart-dsl	0	3	0	0	0	0.3	29.7
mongoBeans	0	4	0	0	6	0.2	51.8
plantuml[1]	0	2	0.6	0	0	0.4	59.6
isis-script	4	15.3	23.1	0	0	14.8	68.2
CheckerDSL	22	4.2	57.9	5	20	35	107.4
Total Count	26	54.5	81.7	5	27	50.8	421.6

[1] To shorten the table, we abbreviate the language name "elite-se.xtext.languages.plantuml" to "plantuml".

5. Results

We now present the final version of the used prompt, and the results obtained from applying our LLM-based solutions to the seven cases.

Finalization of Prompting Text. Following the method described in Step 2, we obtained a version of the prompting text after two rounds of "adjustment-verification" on the example *Domainmodel*, through which we obtained instances from the LLMs (i.e., GPT-3 and Claude-4) that complied with the evolved grammar. We verified this version of the prompt text 10 times for both LLM solutions, and in most of these ten times, the instances obtained complied with the evolved grammar. Therefore, we decided to adopt it as the final version of the prompting text, as follows:

> **Final prompt**: <GRAMMAR_1> is the initial grammar of the DSL. We evolved it to get <GRAMMAR_2>. <INSTANCE_1> was originally a text instance that followed <GRAMMAR_1>. Now I want you to analyze the differences between the two versions of the grammar and, based on these differences, modify <INSTANCE_1> and get <INSTANCE_2>, which will follow <GRAMMAR_2>. Please address the following things:
> 1. When evolving the instance, please do not omit any mandatory elements, such as characters enclosed by single quotes.
> 2. If <GRAMMAR_2> adds a new grammar rule or a new attribute that is optional or in an "OR" relationship (i.e., |), then please do not instantiate it.
> 3. Do not miss or add any auxiliary information in the instance, e.g., comments, formats (white space, indents, tabs, empty lines, etc.).

Compared to the first version, the final version of the prompting text explicitly adds three items that the LLM needs to do. This is based on the problems we encountered in the process of optimizing the prompting text. The first is added because the LLM ignored the symbol ',', which is a mandatory element. The second is added because LLM would actively instantiate the grammar rule "PackageDeclaration" which is a newly added optional rule. The rationale behind this instruction is to ensure minimal changes to the instance, modifying only what is necessary to maintain conformance with the evolved grammar. The third item is added because LLM partially ignored comments.

Co-evolution in Seven Case Languages (RQ1). For each case language, we obtained an generated instance through the two Python scripts and the prompting text, and we repeated this generation operation ten times. We manually compare these ten generated instances with *instance 1* and *grammar 2* one by one to collect data, and then average the data. An overview of the results from Claude-3.5 is presented in Table 3, while the results for GPT-4o are presented in Table 4. In addition, we added the average values on a certain metric vertically, leading to a *total count* – shown in the last row. Based on the results presented in Table 3 and 4, we can now address our first research question regarding how LLMs can be used to automate the co-evolution process.

Table 4
This table shows the results of the GPT-based solution in seven case languages. For each case language, we compare the instance output by GPT-4o with *instance 1* and then count the various metrics of the evolution. We execute the evolution ten runs and then average the ten counts.

Name	#LineErr	#LineEvl	#LineEvlWrg	#LineCmtLost	#LineCmtSave	#LineFmtLost	#LineFmtSave
xtext-orm	13.9	2.2	12.3	0.1	0.9	10.9	57.3
xtext-dnn	1.7	13.5	1.7	0	0	0.1	32.9
smart-dsl	0	3	0	0	0	0	30
mongoBeans	0	4	0	0	6	0.6	51.4
plantuml[1]	2.8	1.9	0.5	0	0	0.6	59.4
isis-script	33.7	3.4	29.5	0	0	8.2	83.6
CheckerDSL	8	0.5	11.6	0	25	0.1	168.7
Total Count	60.1	28.5	55.6	0.1	31.9	20.5	483.3

[1] To shorten the table, we abbreviate the language name "elite-se.xtext.languages.plantuml" to "plantuml".

Answer to RQ1: We developed two automated solutions based on Claude-3.5 and GPT-4o, each consisting of a dedicated Python script and an optimized prompting text. These solutions take the grammar before evolution, the evolved grammar, and the instance that conforms to the grammar before the evolution, then use LLMs to analyze grammar differences and generate an evolved instance that conforms to the evolved grammar.

Correctness evaluation (RQ2). To address RQ2, on the capability of LLMs to produce correct solutions for the underlying co-evolution task, we consider three metrics related to correctness: #LineErr (i.e., count of lines with grammar errors in the evolved instance), #LineEvl (i.e., count of lines of instance 1 that are correctly evolved), and #LineEvlWrg (i.e., count of lines of instance1 that are missing or incorrectly evolved in the evolved instance).

We found that in two DSLs, "smart-dsl" and "mongoBeans", the two LLMs made no mistakes in performing the co-evolution of the instance. In both cases, all lines that needed to change were evolved correctly (see #LineEvl and #LineEvlWrg). The resulting instances included no grammatical errors and, thus, conformed to the new grammar (#LineErr).

In the two cases "xtext-orm" and "xtext-dnn" Claude-3.5 performed better than GPT-4o. Claude-3.5 performed all 10 runs to co-evolution the instance of "xtext-orm" correctly, and only evolved one line incorrectly during one of the 10 runs to co-evolve the instance of "xtext-dnn". Fortunately, the result of this evolution operation still conforms to the evolved grammar. Note that an incorrectly evolved line still conforms to the grammar, e.g., when a line is substituted by an empty line. For "xtext-dnn" GPT-4o produced on average 1,7 lines that were erroneous regarding the evolved grammar and the same number of lines that were evolved wrongly. However, GPT-4o performed not well for "xtext-orm", were it produced on average 13,9 erroneous lines regarding the evolved grammar and 12,3 lines that were evolved incorrectly. Note that an incorrectly evolved line can cause grammatical issues with other lines that have not been changed at all, e.g., when a closing bracket is removed lines that follow might not be parsed as intended. For the language "elite-se.xtext.languages.plantuml" Claude-3.5 managed to always create an instance that conforms to the evolved grammar. However, on average 0,6 lines of the instance were evolved incorrectly over the 10 runs. Here, GPT-4o evolved slightly fewer lines incorrectly (0,5 lines on average), but also failed to systematically create instances that conform to the evolved grammar (with an average of 2,8 erroneous lines).

However, both LLMs made mistakes when performing the co-evolution of the instances for the languages "isis-script" and "CheckerDSL". For example, in the co-evolution of "isis-script", Claude-3.5 evolved on average 23.1 lines incorrectly, while GPT-4o evolved on average 29.5 lines incorrectly. In the co-evolution of "CheckerDSL", GPT-4o outputs better results than Claude-3.5, at least when looking at our metrics. Here Claude-3.5 evolves more than 30 lines incorrectly each time it co-evolves "CheckerDSL" (on average 57,9 lines), producing on average 22 lines that do not conform to the evolved grammar. We compared the instances generated by Claude-3.5 in ten runs with the *instance 1*. We found that in eight of the ten runs, the evolved instance generated by Claude-3.5 for "CheckerDSL"

only contained about the first 140 lines, while *instance 1* had 173 lines. I.e., about the last 30 lines were directly discarded by mistake. In the other two runs, Claude-3.5 did not successfully generate an evolved instance, but only outputted suggestions on how to evolve the instance.

Faced with such differences between the different DSLs, we looked further into the size of the instances in those languages. We found that three languages, "CheckerDSL", "isis-script", and "xtext-orm", have larger instances. From the evolution results of the two LLMs in the seven case languages, the errors also mainly appear in these three languages. Thus, it seems that the size of the instances is a factor that might affect the performance of the LLMs when executing the co-evolution.

Answer to RQ2: The LLMs performed excellently on small instances (such as those of "smart-dsl" and "mongoBeans"), correctly executing all necessary evolution operations. However, their performance significantly degraded when co-evolving larger instances (such as those of "isis-script" and "CheckerDSL"). We conclude that the instance size affects the correctness of LLM-generated solutions.

Support for auxiliary information (RQ3). We consider how well the LLMs performed in preserving auxiliary information, in terms of the metrics #LineCmtLost and #LineCmtSave (i.e., count of lines of instance 1 with comments that are lost and retained, respecitvely), as well as #LineFmtLost and #LineFmtSave (i.e., count of lines of instance 1 with format information lost and retained, respectively).

Preservation of Comments. Only the instances of "xtext-orm", "mongoBeans", and "CheckerDSL" contained comments. In the co-evolution of grammar and instances in language "mongoBeans" and "xtext-orm", both LLMs successfully preserved all comments (only in one co-evolution run of "xtext-orm", GPT-4o lost a comment). However, when evolving "CheckerDSL", GPT-4o retained comments better than Claude-3.5. As mentioned in Section 5, Claude-3.5 did not generate an evolved instance in two of the ten evolution runs for "CheckerDSL". As a side effect, all comments in *instance 1* were lost because no evolved instance was generated.

Preservation of Formats. From the data, GPT-4o lost only a few lines of formatting information when performing the co-evolution. However, when we further opened the instance files, we found that this was not the case. Sometimes GPT-4o does not perform the evolution that should occur during some co-evolution executions, but directly copies the text content of the *instance 1* to the new instance. For example, in seven out of ten runs of the co-evolution in "CheckerDSL", GPT-4o directly copied all the text content of *instance 1*. I.e., the text of the entire instance was not modified, so the original formatting information was completely copied to the evolved instance. We found that in case languages with a small number of lines of instance text (e.g., "mongoBeans"), LLM can well preserve formatting information during co-evolution.

Answer to RQ3: In terms of preserving auxiliary information, both LLMs performed well in maintaining comments and formatting information when handling small instances. However, GPT-4o would sometimes directly copy the original instance without performing necessary evolution operations. For larger instances (such as instance of CheckerDSL), Claude-3.5 exhibited output truncation issues, resulting in the loss of comments and formatting information. This indicates that LLMs' capability to preserve auxiliary information is significantly affected by instance size.

6. Discussion

In general the initial results of both LLMs are promising, as they indicate that, at least for small DSLs, evolution steps, and instances, a co-evolution while maintaining auxiliary information is possible. In the following, we discuss some open issues and threats to validity.

Scalability. Our results indicate that the good initial results do not scale well for larger textual instances, larger grammars, and more significant changes between grammar versions. While this

does not invalidate LLMs' applicability to practical cases as long as they remain small—DSLs are often conceived as "small languages" for dedicated tasks [21]—applying an LLM-based solution to large practical cases is currently challenging. Future work should explore approaches to improve scalability, including techniques for handling larger inputs and potentially generating migration programs rather than performing direct co-evolution. As more powerful LLMs become available, their capability to handle larger instances during co-evolution will likely increase.

Non-grammar-driven Instance Changes. While we identified instances in repositories that conform to evolved grammars, these instances often contain changes unrelated to grammar evolution. For example, in "isis-script", numerous action objects were deleted between versions, though this was not required for grammar conformance. This makes it difficult to use real updated instances as a baseline for evaluating whether LLMs perform co-evolution similarly to human developers. Future work should explore ways to distinguish between pure co-evolution changes and other modifications developers make during language evolution.

Variations in Migration Strategies. Research on meta-model to model co-evolution has shown that there is sometimes more than just one possible and valid outcome of a migration [5]. The same is likely true for textual instances. Therefore, we would like to address this issue in future work, developing an approach to consider different migration strategies. Alternatively, expanding on the idea to use LLMs to generate migration programs, one could also use LLMs to generate configurable migration code.

Threats to Validity. The primary limitation of this study is the relatively small set of case languages and their associated instances, which may not fully represent the diversity of DSLs in practice. Additionally, the instances found in repositories were likely designed for demonstration purposes rather than production use, potentially overestimating LLMs' performance on real-world cases. Finally, Claude-3.5's consistent output truncation for CheckerDSL (approximately 170 lines) prevented us from fully evaluating its capabilities on larger instances, suggesting that our findings on scalability should be interpreted cautiously until further investigation with improved techniques for handling larger inputs. We derived our prompt from a single case language, which may lead to overfitting. To mitigate this threat, we applied our approach to seven diverse case languages selected from different domains with varying complexity levels, and performed ten runs for each case language to account for LLM output variability. The observed performance variations across these cases provide insights into the generalizability limitations of our approach. In future work, employing cross-validation techniques using multiple representative DSLs during prompt development could further reduce this threat.

7. Conclusion

This paper explored the potential of using LLMs (specifically Claude-3.5 and GPT-4o) to support co-evolution between DSL grammar definitions and instances. Our experiments across seven case languages demonstrated that LLMs can effectively handle co-evolution tasks while preserving auxiliary information like comments and formatting, particularly for smaller instances However, performance degraded with larger, more complex languages and significant grammar changes. Key limitations included truncated output for large instances and inconsistent handling of complex grammar modifications.

We foresee several directions for future work. We plan to extend our research to graphical DSLs to examine LLMs' effectiveness across different modeling paradigms. Moreover, we will conduct a comparative analysis between our LLM-based approach and manual co-evolution processes to quantify potential benefits in practical scenarios. We also consider further research on prompt engineering, particularly exploring the impact of specific versus general terms in identifying and retaining auxiliary information, and investigating LLMs' sensitivity to grammar rule ordering in formal language definitions. Additionally, we intend to develop an enhanced evaluation framework with precise quantitative metrics. Moreover, we hope to systematically evaluate the quality of newly added information in co-evolved instances, providing deeper insights into LLMs' capabilities for supporting comprehensive DSL evolution.

References

[1] R. Lämmel, Software Languages, Springer, 2018.

[2] T. Martin, B. Azvine, Evolution of fuzzy grammars to aid instance matching, in: 2006 International Symposium on Evolving Fuzzy Systems, IEEE, 2006, pp. 163–168.

[3] T. Martin, Y. Shen, B. Azvine, Incremental evolution of fuzzy grammar fragments to enhance instance matching and text mining, IEEE Transactions on Fuzzy Systems 16 (2008) 1425–1438.

[4] S. Vaupel, D. Strüber, F. Rieger, G. Taentzer, Agile bottom-up development of domain-specific ides for model-driven development, in: Workshop on Flexible Model Driven Engineering, CEUR-WS.org, 2015, pp. 12–21.

[5] R. Hebig, D. E. Khelladi, R. Bendraou, Approaches to co-evolution of metamodels and models: A survey, IEEE Transactions on Software Engineering 43 (2016) 396–414.

[6] M. Latifaj, F. Ciccozzi, M. Mohlin, E. Posse, Towards automated support for blended modelling of uml-rt embedded software architectures., in: ECSA (Companion), 2021.

[7] J. Holtmann, J.-P. Steghöfer, W. Zhang, Exploiting meta-model structures in the generation of xtext editors., in: MODELSWARD, 2023, pp. 218–225.

[8] B. Yang, Z. Liping, Z. Fengrong, A survey on research of code comment, in: Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences, 2019, pp. 45–51.

[9] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, B. Myers, Using an llm to help with code understanding, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.

[10] Y. Dong, X. Jiang, H. Liu, Z. Jin, G. Li, Generalization or memorization: Data contamination and trustworthy evaluation for large language models, arXiv preprint arXiv:2402.15938 (2024).

[11] L. Bettini, Implementing domain-specific languages with Xtext and Xtend, Packt, 2016.

[12] W. Zhang, D. Strüber, Tales from 1002 repositories: Development and evolution of xtext-based dsls on github, in: 2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2024, pp. 172–179.

[13] W. Zhang, J. Holtmann, D. Strüber, R. Hebig, J.-P. Steghöfer, Supporting meta-model-based language evolution and rapid prototyping with automated grammar transformation, Journal of Systems and Software 214 (2024) 112069.

[14] W. Zhang, J.-P. Steghöfer, R. Hebig, D. Strüber, A rapid prototyping language workbench for textual dsls based on xtext: Vision and progress, arXiv preprint arXiv:2309.04347 (2023).

[15] D. Wagelaar, L. Iovino, D. Di Ruscio, A. Pierantonio, Translational semantics of a co-evolution specific language with the emf transformation virtual machine, in: Theory and Practice of Model Transformations: 5th International Conference, ICMT 2012, Prague, Czech Republic, May 28-29, 2012. Proceedings 5, Springer, 2012, pp. 192–207.

[16] A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio, Automating co-evolution in model-driven engineering, in: 2008 12th International IEEE enterprise distributed object computing conference, IEEE, 2008, pp. 222–231.

[17] L. M. Rose, D. S. Kolovos, R. F. Paige, F. A. Polack, S. Poulding, Epsilon flock: a model migration language, Software & Systems Modeling 13 (2014) 735–755.

[18] J.-P. Tolvanen, S. Kelly, J. Di Rocco, A. Pierantonio, G. Tinella, A framework for evaluating tool support for co-evolution of modeling languages, tools and models, Software and Systems Modeling (2024) 1–28.

[19] J. Di Rocco, D. Di Ruscio, C. Di Sipio, P. T. Nguyen, R. Rubei, On the use of large language models in model-driven engineering, Software and Systems Modeling (2025) 1–26.

[20] Z. K. Kebaili, D. E. Khelladi, M. Acher, O. Barais, An empirical study on leveraging llms for metamodels and code co-evolution, in: European Conference on Modelling Foundations and Applications (ECMFA 2024), volume 23, Journal of Object Technology, 2024, pp. 1–14.

[21] A. V. Deursen, P. Klint, Little languages: little maintenance?, Journal of Software Maintenance: Research and Practice 10 (1998) 75–92.