CHALMERS
UNIVERSITY OF TECHNOLOGY

Creating Python-Style Domain Specific Languages: A Semi-Automated Approach and Intermediate Results

Creating Python-Style Domain Specific Languages: A Semi-Automated Approach and Intermediate Results

Weixing Zhang[1][a], Regina Hebig[1][b], Jan-Philipp Steghöfer[2][c] and Jörg Holtmann[1][d]
[1]*Department of Computer Engineering, Chalmers University of Technology, University of Gothenburg, Sweden*
[2]*Xitaso IT & Software Solutions GmbH, Augsburg, Germany*
{*weixing.zhang, regina.hebig, jorg.holtmann*}*@gu.se, jan-philipp.steghoefer@xitaso.com*

Keywords: DSL, Xtext, Textual Modeling, Grammar, Language Engineering.

Abstract: Xtext is a well-known domain-specific language design framework and technology. It automatically generates a textual grammar for a language, given a meta-model specified in Ecore. These generated textual grammars are typically not user-friendly. Python-style languages are popular among developers for their usability and conciseness. We aim to propose a systematic approach to transform a DSL with a generated grammar into a Python-style DSL. To achieve this, we analyze the problems of grammars generated with Xtext, based on a lightweight architecture description language. In response to these problems, we propose a general semi-automated grammar adaptation approach. We apply the approach to two other DSLs to validate the generalization of the approach. We also discuss the limitations of this approach and prospects for the future.

1 INTRODUCTION

In contrast to general-purpose programming languages (GPLs) like Java, domain-specific languages (DSLs) are computer languages tailored for a particular application domain (Kosar et al., 2016). DSLs come in a variety of forms and are employed in a wide range of domains(do Nascimento et al., 2012). For example, DSLs are used to improve the level of abstraction and automation in the application development in the robotic domain (Nordmann et al., 2014). But it is important to take into account the workload that goes into creating the DSL (Mernik et al., 2005). The good news is that there are many tools available for designing and developing DSLs, like JetBrains MPS, MontiCore, Xtext, Racket, etc. (Iung et al., 2020). One of them is Xtext (Eclipse Foundation, 2022c), an open-source software framework that simplifies the development of DSLs. Xtext can generate a complete DSL infrastructure, including parsers, linkers, compilers, etc. Developers utilize Ecore meta-models to represent domain ideas and their relationships when creating DSLs based on Xtext. From such a meta-model, Xtext generates grammar that specifies the concrete syntax. The editor and its different components are automatically generated from the meta-model of this language as Xtext artifacts.

The grammar generated by Xtext tends to specify languages that are not user-friendly to use. For example, the generated DSLs include default requirements that developers type in all keywords, use braces to an extensive amount, etc. This leads to a high effort in writing when programming in these DSLs. In contrast, the Python (Python Community, 2022) language provides a very clean coding style, is renowned for increasing programmer productivity, and is considered easy to learn (Gmys et al., 2020). Both are properties are also relevant and desirable for DSLs. However, there is currently no systematic approach for converting DSLs to Python-like languages.

In this paper, we introduce an approach to semi-automatically convert textual DSL grammars generated by Xtext to a Python-like DSL. This means that the resulting languages will be easy to code, easy to read, user-friendly, concise, and use whitespace and indents instead of braces (Gmys et al., 2020). We developed the approach by developing an architectural description language (DemoAdl) as an exemplar for automatically transforming a DSL to a Python-like language. Once our transformation worked for DemoAdl, we evaluated the approach with additional DSLs, *Xenia* and *ACME*. This way we validate the

[a] https://orcid.org/0000-0003-2890-6034
[b] https://orcid.org/0000-0002-1459-2081
[c] https://orcid.org/0000-0003-1694-0972
[d] https://orcid.org/0000-0001-6141-4571

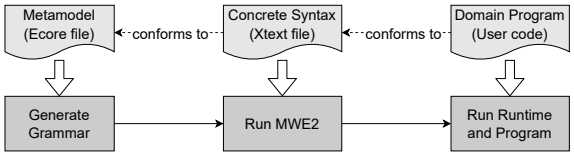

Figure 1: The relationship between different artifacts in the Xtext-based MDSE solution.

generalizability of our approach and explore its limitations.

2 BACKGROUND

As outlined in the introduction, Xtext is a framework for developing DSLs, and Xtext-based Model-Driven Software Engineering (MDSE) is a common solution for developing DSLs (Behrens et al., 2008). In this solution, the DSL developer uses an Ecore meta-model to describe the concepts of the domain and the relationships between the concepts (Steinberg et al., 2008). For example, "port" becomes a class in the meta-model when using a meta-model to describe concepts in the field of system architecture. Next, the concrete syntax is generated from the meta-model by using the Xtext framework. Xtext provides Modeling Workflow Engine 2 (MWE2) (Eclipse Foundation, 2022a), a declarative, externally configurable generator engine. After having a concrete syntax, all Xtext artifacts can be generated by running the MWE2 file. These artifacts actually make up the editor for the DSL, including linker, parser, type-checker, etc., common components of editors, and editing support for Eclipse. After the editor is generated, code written in the DSL could be resolved and supported at runtime. Figure 1 shows the relationship between the above concepts. When Xtext generates a textual grammar, it will generate a corresponding grammar rule for each class or enumeration in the meta-model and create an attribute in the textual grammar for each attribute or association (i.e., reference or containment) in the meta-model. A grammar rule corresponding to a class may contain multiple attributes, and each attribute occupies a line of text. A grammar rule corresponding to an enumeration contains multiple alternative values, usually on the same line of text.

3 METHODOLOGY

As mentioned above, we developed the approach by developing an architectural description language (i.e., DemoADL) for an example embedded system. The DSL can be used to describe simple embedded system structures, which include hardware components and software components. There are different types of hardware components, and each software component contains a number of sub-components. There are direct links between hardware components and software components, which together make up the system.

example.dadl

```
softwarecomponent {
    SoftwareComponent {
        identifier swcomp1
        port {
            FunctionPort {
                identifier funcPort1
                direction in
            }
        }
        portgroup {
            PortGroup {
                identifier portgroup1
                port {
                    PowerPort {
                        identifier powerPort1
                        inputVoltage 11
                    },
                    PowerPort {
                        identifier powerPort2
                        inputVoltage 12
                    }
                }
            },
            PortGroup {
                identifier portgroup2
                port {
                    PowerPort {
                        identifier powerport2
                        inputVoltage 13
                    },
                    FunctionPort {
                        identifier funcPort3
                        direction out
                    }
                }
            }
        }
        connector {
```

Figure 2: A screenshot of part of the domain program with a default Xtext style.

We created an Ecore meta-model that describes the domain concepts from this system and generated textual grammar from the meta-model by using Xtext. This constitutes the default grammar for the concrete syntax for DemoAdl. We used this DSL to code an example program with a default style (we called this program a "Default-style program"). We analyzed the shortcomings of the grammar and the program conforms to it. We also created a draft of the program to illustrate what it might look like in a Python-style grammar. We call this program a "Python-like draft".

Based on that analysis, we developed a systematic approach to adapt grammars that were generated with Xtext. The approach is based on grammar adaptation rules in response to the problems analyzed from the default-style DSL. We applied these rules to the grammar Xtext generated for DemoAdl. To semi-automate the grammar adaptations, we developed a script in the form of an eclipse Java plug-in. After the adaptation of the grammar, we generated an editor and wrote the same program according to the newly adapted grammar (we called this code a "Python-like program"). This was to test whether we could successfully parse

such a program in the new editor.

To evaluate the generalizability of our approach, we chose two other DSLs. Our selection criteria are that the language 1) has an accessible homepage and 2) with examples on the homepage, and 3) has a downloadable Ecore meta-model. We apply the proposed approach to these two languages to determine whether the script and approach can be applied to create Python-style languages.

4 RESULTS

We present our analysis, our semi-automated approach, and an evaluation of our approach.

4.1 Analysis

As mentioned in the methodology section, we first analyzed the shortcomings of the sample DSL (i.e., DemoADL). We illustrate our findings with the help of a snippet of the default style program in Figure 2. We made the following observations about the grammar generated by Xtext:

1. **Inappropriate Position of Identifier.** In Xtext, 'name' is the default name for an element's identifier. However, when we use an attribute named *identifier* to identify an element (e.g., a software component in our case language DemoAdl), then it will default to a normal attribute and be placed within braces. This means that, for example, when we type in keyword *SoftwareComponent*, we have to type in the left brace first and then the attribute identifier. For example, in Figure 2 (line 22), attribute *identifier* with value 'swcomp1' is used to identify an element *SoftwareComponent*, however, the attribute *identifier* is existing inside of braces in the second line which reduces the brevity of the code. If the same content is expressed in Python, e.g., in Figure 3, the identifier value 'swcomp1' (line 1) will follow the keyword *SoftwareComponent* to identify the element.
2. **Heavy Separation of Code Blocks.** The default-style program uses both braces and commas to separate and distinguish code blocks. For example, *PortGroup* 'portgroup1' and 'portgroup2' are on the same level while they are separated by a comma symbol. Obviously, these commas are redundant, because braces have already been able to separate and distinguish them. In Python, for example in Figure 3, 'portgourp1' and 'portgroup2' are separated by being on different lines though there is no brace or comma for them. This is because there are indents to express hierarchy, and 'portgourp1' and 'portgroup2' are on the same hierarchy level.
3. **Repetitive Keyword.** There are different keywords with the same functional implication. For example, there are two keywords *softwarecomponent* and *SoftwareComponent*, which are highly redundant in function. This also reduced the usability of the language.
4. **Nested Braces.** There are many nested braces, for example in Figure 2, after typing in a keyword, it's necessary to open a brace. Next, we should type in the keyword *PortGroup* and go into a brace again. These nesting braces are unnecessary and redundant, which increases the complexity of the code. In Python, braces are avoided by having a whitespace-sensitive syntax.

```
1  SoftwareComponent swcomp1
2      FunctionPort funcPort1 direction in
3      PortGroup portgroup1
4          PowerPort powerPort1 inputVoltage 11
5          PowerPort powerPort2 inputVoltage 12
6      PortGroup portgroup2
7          PowerPort powerport2 inputVoltage 13
8          FunctionPort funcPort3 direction out
9      PortConnector
10         ports (funcPort1, "swcomp1.portgroup1.powerPort1")
11     GroupConnector
12         portgroups (portgroup1, portgroup2)
13
```

Figure 3: Snippet of `SoftwareComponent` in the system architecture description example, in Python style.

The draft shown in Figure 3 shows how the program from Figure 2 could appear in a Python-style language. The envisioned code in Figure 3 uses whitespace and indentations to define and separate code blocks, which greatly improves the conciseness of the code. Identifier of e.g. *SoftwareComponent* directly follows after the keyword 'SoftwareComponent' (cf. line 1 in Figure 3), which makes the code more readable. There are no more braces and the number of keywords and symbols is greatly reduced.

4.2 The Semi-Automated Approach

As mentioned earlier, we developed a semi-automated approach to adapt a generated grammar to become more like Python. On the one hand, we develop a script that applies some changes automatically. On the other hand, we require the language engineer to take some specific decisions that must be done by hand. The adaptations that are automatically completed by the script are removing braces, repositioning attributes, and removing commas. Hand-crafted adaptations address which keywords need to be refined. In the following we describe these adaptations in more detail:

Remove Braces and Introduce White-Space Awareness. Removing all braces directly from the grammar definition may cause errors, such as left recursion errors(Bettini, 2019), which will prevent the creation of a textual editor.

Therefore, the functionality of the braces needs to be substituted with the use of whitespace and indents to define code blocks. Thus, we need the language to be whitespace-aware (Eclipse Foundation, 2022b). To this end, the script introduces the following changes:

Import Required Features. Change the reference in the grammar definition file from *org.eclipse.xtext.common.Terminals* to *org.eclipse.xtext.xbase.Xbase*. In addition, import *Xbase* to refer to *EClassifiers* from that model. This gives access to features that allow white-space-aware grammars in Xtext.

Create BEGIN/END Terminals. Include whitespace-aware blocks in your language by using synthetic tokens in the grammar of the form '`synthetic:<terminal name>`'. An example using *BEGIN*/*END* is shown in Figure 4.

Redefine Expression. Inherits expressions from *Xbase* and redefines the syntax of block expressions by overriding the definition of *xbase::XExpression*.

Reposition Attribute Identifier. With the help of the script, we moved the attribute *identifier* from its original position to after the keyword with the same name as the grammar rule. For example, *SoftwareComponent* would be exactly followed by the attribute *identifier*. Here, the script uses regular expressions to find the attribute identifier and move it. If the identifier is called *name*, we do not need to do anything with it.

Remove Commas. In this step, we removed commas that separate code blocks. For example, if there are multiple *port*s under the same *PortGroup* (cf. line 11 & 12 in Figure 5), we do not need to separate these ports with commas because whitespace and indentations are used instead.

Refine Keywords. In this step, functionally redundant keywords are manually removed. The aforementioned keywords *SoftwareComponent* and *softwarecomponent*, e.g., were functionally redundant and we removed one of them (in our case, we kept the upper case one). We did not implement this removal in the script, because the keywords are language-specific and people make different decisions about what to keep. At the same time, we also did not implement the removal of the *BEGIN*/*END* related to them in the script. We will address these two automated operations in our future work by configurable rules with a finite set of options. For manually removing functionally redundant keywords, we recommend defining a rule, e.g., to remove the keyword which is written entirely in lowercase. However, our script automatically removes the keyword 'identifier', because, in our envisioned draft, the value of the *identifier* should directly follow the keyword (e.g., *SoftwareComponent*) without the existence of the keyword 'identifier', which would make the code more concise.

When the script adapts the grammar, it uses regular expressions to search for the target texts in the entire grammar text, and then performs operations on it, including deletion, modification, etc.

4.3 Evaluation

With the above modifications, the grammar of DemoADL was changed. We generated the Xtext artifacts for the language by running the MWE2 workflow file. We wrote a program (cf. Figure 5) that conforms to the newly adapted textual grammar, which was parsed successfully by the generated editor.

To evaluate the usability and generalizability of the proposed approach, we applied it to two additional DSLs, *Xenia* and *ACME*, which we selected based on the criteria described above. The basic information of the two DSLs is shown in Table 1.

Grammar Generation with Xtext. The Ecore meta-model of ACME is from the Atlantic Zoo (AtlanMod, 2019). To fulfill all technical constraints necessary for the generation of model code and grammar with Xtext, some small adaptations were necessary:

1. We filled in the namespace values (i.e., 'Ns Prefix' and 'Ns URI') for all packages in the meta-model.
2. We filled in the value (i.e., 'Instance Type Name' for all the types under the package *primitivetypes*.
3. We changed the lower-bound value of two attributes under the type *Link* from 1 to 0.

We then generated a textual grammar from both DSLs' meta-models using the Xtext framework.

Applying the Approach. With the proposed approach, we adapted both DSLs to a Python-like DSL with the help of our script. All manual steps were performed by the first author of this paper. For both *ACME* and *Xenia*, executing these manual steps took less than 30 minutes each.

```
SoftwareComponent returns SoftwareComponent:
    {SoftwareComponent}
    'SoftwareComponent'
    '{'
        ('identifier' identifier=EString)?
        ('CompID' CompID=EString)?
        ('allocatedTo' allocatedTo=[Node|EString])?
        ('port' '{' port+=Port ( "," port+=Port)* '}' )?
        ('portgroup' '{' portgroup+=PortGroup ( "," portgroup+=PortGroup)* '}' )?
        ('connector' '{' connector+=Connector ( "," connector+=Connector)* '}' )?
    '}';
```

(a) Grammar rule SoftwareComponent before adaptation.

```
SoftwareComponent returns SoftwareComponent:
    {SoftwareComponent}
    'SoftwareComponent'
    identifier=EString
    BEGIN
        ('CompID' CompID=EString)?
        ('allocatedTo' allocatedTo=[Node|EString])?
        (port+=Port (port+=Port)*)?
        (portgroup+=PortGroup (portgroup+=PortGroup)*)?
        (connector+=Connector (connector+=Connector)*)?
    END;
```

(b) Grammar rule SoftwareComponent after adaptation.

Figure 4: Comparison of grammar rules SoftwareComponent before and after adaptation.

Table 1: The two languages chosen for the evaluation.

Language	Domain	Language elements	meta-model Source	Homepage
Xenia	Generate web pages	14	(Rodchenkov, 2019)	(Rodchenkov, 2020)
ACME	SW Architecture description	16	(AtlanMod, 2019)	(ABLE Group, 2011)

example.dadl

```
System
    hardwarecomponent
        Actuator run
        Node process
            SoftwareComponent swcomp2
                CompID huhjlj
        Sensor sensor1
    SoftwareComponent swcomp1
        FunctionPort funcPort1 direction in
        PortGroup portgroup1
            PowerPort powerPort1 inputVoltage 11
            PowerPort powerPort2 inputVoltage 12
        PortGroup portgroup2
            PowerPort powerport2 inputVoltage 13
            FunctionPort funcPort3 direction out
        PortConnector
            ports (funcPort1, "swcomp1.portgroup1.powerPort1")
        GroupConnector
            portgroups (portgroup1, portgroup2)
```

Figure 5: Screenshot of part of the example program for the DemoADL language with a Python-like style in eclipse.

Outcome. An example program for the grammar that Xtext generated for *Xenia* is shown in Figure 6-(b) while the program for the resulting Python-like grammar of *Xenia* is shown in Figure 6-(c). For comparison, two programs correspond to the "app Main" example from the *Xenia* home page (Rodchenkov, 2020) (shown in Figure 6-(a)).

The comparison shows that the program in (c) is much more concise than the one in (b) because there are fewer keywords and nesting. Like Python, the program in (c) uses whitespace and indents to express hierarchy. The comparison to the original program (Figure 6-(a)) also shows that the resulting Python-like grammar is much closer in terms of compactness to the actual, intended grammar of Xenia.

An example program for the grammar that Xtext generated for *ACME* is shown in Figure 8 while the program for the resulting Python-like grammar of *ACME* is shown in Figure 9. Similarly, the two programs conform to the example "simple_cs" from the subpage "An Overview Of Acme" of the *ACME* homepage (ABLE Group, 2011) (shown in Figure 7).

Again we can see that the Python-like program in (c))contains much fewer lines of code, and there is no need to input braces in the program. Every identifier (here e.g., the name) follows the main keyword to identify a certain structure. Also, the comparison to the original grammar shows that the Python-like style is much closer to the original and intended grammar of *ACME* (Figure 7) in terms of compactness.

Note that the original syntaxes of both *Xenia* and *ACME* are quite different in style and neither is originally white-space sensitive. This shows that our approach and the script are applicable to diverse DSLs and can be used by DSL developers to quickly reach a Python-like grammar, which could then be used as a basis for further refinements of the grammar.

5 DISCUSSION

5.1 Threats to Validity and Limitations

Threats to Validity. As discussed in the evaluation part, we applied the proposed approach to the two DSLs Xenia and ACME. The results show that the approach could be successfully used for these two DSLs. However, to ensure generalizability, we plan to apply the approach to additional languages. Although we could show that our method provides a quick way to adapt diverse Xtext-generated grammars to Python-style languages, there are still some limitations.

Expressiveness of Language. Even after adapting a grammar with our approach, there is often room for further modifications to the grammar to improve the expressiveness of the language itself. In the Xenia

```
app Main has pages[
    @Home with modal(@Notification, @Login),
    @Contact with modal(
        @Message,
        @Logout with modal(
            @Contact
        )
    ),
    @Message
]
with: "native"
mode: DEV, // or PROD
xml: "/home/user/foliage/map.xml"
map :[
    Message -> (Message),
    Login -> (Home, Contact)
]
```

(a) The example from Xenia official website.

example.defaultxenia

```
Model {
    headers {
        Header {
            appName Main
            sites {
                SiteWithModal Home {sites
                    {Site Notification, Site Login}
                },
                SiteWithModal Contact {
                    sites {
                        Site Message,
                        SiteWithModal Logout {
                            sites {
                                Site Contact
                            }
                        }
                    }
                },
                Site Message
            }
        }
    }
    entities {
        Entity { tech "native" },
        Entity { mode DEV },
        Entity { path "/home/user/foliage/map.xml" }
    }
    mapped_entities {
        MappedEntity {
            linkedProps {
                LinkedProperty {
                    name Message page RedirectPage
                    { site (Message) }
                },
                LinkedProperty {
                    name "Home.Login" page RedirectPage
                    { site (Message, "Contact.Logout.Contact") }
                }
            }
        }
    }
}
```

(b) The same Xenia example according to grammar generated by Xtext.

example.pythonicxenia

```
headers
    appName Main
    sites
        Home
            Notification
            Login
        Contact
            Message
            Logout
                Contact
        Message
entities
    tech "native"
    path "/home/user/foliage/map.xml"
mapped_entities
    linkedProps
        name Message page (Message),
        name "Home.Login" page (Message, "Contact.Logout.Contact")
```

(c) The same Xenia example according to Python-like grammar.

Figure 6: Comparison of Xenia programs in different styles (The original example is from https://github.com/rodchenk/xenia).

```
System simple_cs = {
    Component client = { Port send-request; };
    Component server = { Port receive-request; };
    Connector rpc = { Roels { caller, callee}};
    Attachments {
        client.send-request to rpc.caller;
        server.receive-request to rpc.callee;
    }
}
```

Figure 7: ACME original syntax from https://www.cs.cmu.edu/~acme/docs/language_overview.html.

example, we can change the keyword *page* to the arrow -¿, indicating a “progressive” or “link” relationship as it was done in the original grammar. Such special keywords are typical for DSLs. Our proposed approach does not include such operations, since they are highly language-specific. We focus currently only on adapting a DSL with a default Xtext-generated grammar to Python style. However, in future work, it might be interesting to support such operations with automation tools.

Automation of Grammar Modification. We implemented a small script in the form of an Eclipse plugin to simplify grammar modification. However, the functionality of the script is limited. For example, adding a colon *:* after a certain type of keyword is not

example.myacme

```
System {
    name simple_cs
    componentDeclaration {
        ComponentInstance {
            name client
            instanceOf "="
            ports {
                Port { name "send-request" }
            }
        },
        ComponentInstance {
            name server
            instanceOf "="
            ports {
                Port { name "receive-request" }
            }
        }
    }
    connectorDeclaration {
        Connector {
            name rpc
            roles {
                Role { name caller },
                Role { name callee }
            }
        }
    }
    attachments {
        Attachment {
            comp client port "send-request" con rpc role caller
        },
        Attachment {
            comp server port "receive-request" con rpc role callee
        }
    }
}
```

Figure 8: The example from Figure 7 in the syntax automatically generated by Xtext.

supported by the script at present. A feasible solution will be to extend the functionality to support various

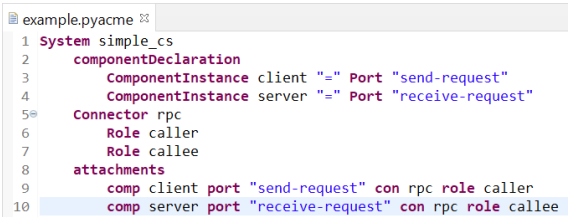

Figure 9: The example from Figure 7 in the Python-like syntax after our tool modified the generated grammar.

modifications of keywords.

Also, while adapting the grammar, one may encounter problems such as *left-recursion* errors. Other than the focused change to white-space awareness, we currently provide no additional support to help developers avoid changes that lead to these errors.

5.2 Future Work

As we mentioned above, in the future we will apply our approach to more languages to evaluate generalizability. In addition, we plan to apply the approach to larger DSLs, for example, EAST-ADL. We hope that this will also help us refine our approach and extend the capabilities of the script. Namely, we hope that we can address the previously mentioned limited functionality of our script and the semi-automated nature of our approach.

Converting a DSL generated from a meta-model into a Python-like language can improve the language's simplicity, friendliness, and usability. However, these are all issues on the language appearance level. In the future, we will discuss and explore how to design a good DSL, including defining the standards of what a good DSL is.

Another interesting and related topic is blended modeling (Addazi and Ciccozzi, 2021), which refers to modeling in different notations (such as textual, graphical, etc.) at the same time. When creating a textual language for a DSL that already has a non-textual notation, the approach in this paper can help create a concise, user-friendly textual language in the first stage, thereby improving the user experience of modeling activities (David et al., 2022). In future work, we plan to evaluate our approach in this context.

6 RELATED WORK

Since the first version of Xtext was released in 2006 (Efftinge and Völter, 2006), the German company *itemis* released several versions of Xtext. Eysholdt and Behrens from the company briefly introduced the motivation and capabilities of Xtext in (Eysholdt and Behrens, 2010). In addition, *itemis* also officially released the Xtext User Guide, which introduces in detail what Xtext is, how to use it and some of its internal principles (Behrens et al., 2008). This is the technical manual that is the main reference for this paper. Bettini's book (Bettini, 2016) goes further and shows how to develop DSLs using Xtext and Xtend. Moreover, the book has a dedicated and independent chapter to introduce Xbase, which is not included in (Behrens et al., 2008) and it supplements the knowledge about Xbase for us.

Mernik et al. in (Mernik et al., 2005) provide empirical data and valuable experience on when and how to develop DSLs, this study refers to these experiences when designing the DSL. Neubauer et al. customized the grammar generated from the Ecore meta-model (Neubauer et al., 2015). However, their purpose was to solve the problem that the Xtext grammar generator did not create rules for meta-model data types. In contrast, the modification in this paper aims to bring the grammar closer to Python. Manual changes were made by Sredojevi to the DSL's grammar definition file in (Sredojević et al., 2015). By including keywords and identifiers, this change primarily addresses the issue that the graphical representation of the meta-model does not fully define the grammar structure. In contrast, our approach replaces the language's overall style by modifying the entire grammar definition file.

7 CONCLUSION

Xtext is one of the most commonly used methods for developing DSL. It can be used to generate a textual grammar from an Ecore meta-model. However, this auto-generated grammar has a format that is often considered cumbersome and difficult to work with. Python is a language known for its simplicity, which helps programmers become more productive. We aim to make it easy for DSL developers to create their languages with a Python-style syntax, in the hope that these DSLs would benefit from the advantages of Python's syntax. In this paper, we analyze the primary inadequacies of an auto-generated DSL (i.e., DSL generated by Xtext) which is to describe the architecture of an embedded system. Based on this analysis, this paper presents an approach to semi-automatically change a grammar that was generated with Xtext, so that the DSL becomes a Python-like language. We applied this approach to the design of the aforementioned lightweight example language and obtained an intermediate result, a Python-like structure description language. We applied the approach to two ad-

ditional DSLs, *Xenia* and *ACME*, to validate the generalizability of our approach. The contribution of this paper is a generalized approach for transforming generated DSLs into Python-like DSLs.

We intend to investigate the practical applications of this approach in the future, e.g., the adaptation of highly sophisticated and large DSLs in industrial settings. In our future efforts, we intend to develop a more complete system for the adaptation of automatically generated grammar. We envision a rule-based system that allows an engineer to configure a number of modifications of the grammar that, together, change the concrete syntax significantly. We believe such a system will contribute to making Xtext more attractive to language engineers who want to combine work on an evolving and rapidly changing language with a user-friendly and compact concrete syntax, a combination that Xtext currently does not support.

REFERENCES

ABLE Group (2011). Acme homepage. At Carnegie Mellon University, https://acme.able.cs.cmu.edu/docs/language_overview.html, Last accessed Nov 2022.

Addazi, L. and Ciccozzi, F. (2021). Blended graphical and textual modelling for uml profiles: A proof-of-concept implementation and experiment. *JSS*, 175:110912.

AtlanMod (2019). Atlantic zoo. https://github.com/atlanmod/atlantic-zoo, Last accessed Nov 2022.

Behrens, H., Clay, M., Efftinge, S., Eysholdt, M., Friese, P., Köhnlein, J., Wannheden, K., and Zarnekow, S. (2008). Xtext user guide. http://www.eclipse.org/Xtext/documentation/, Last accessed Nov 2022.

Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

Bettini, L. (2019). Type errors for the ide with xtext and xsemantics. *Open Computer Science*, 9(1):52–79.

David, I., Latifaj, M., Pietron, J., Zhang, W., Ciccozzi, F., Malavolta, I., Raschke, A., Steghöfer, J.-P., and Hebig, R. (2022). Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study. *Software & Systems Modeling To appear*.

do Nascimento, L. M., Viana, D. L., Neto, P., Martins, D., Garcia, V. C., and Meira, S. (2012). A systematic mapping study on domain-specific languages. In *ICSEA*, pages 179–187.

Eclipse Foundation (2022a). Mwe2. https://www.eclipse.org/Xtext/documentation/306_mwe2.html. Last accessed Nov 2022.

Eclipse Foundation (2022b). Whitespace-awareness. https://www.eclipse.org/Xtext/documentation/307_special_languages.html. Last accessed Nov 2022.

Eclipse Foundation (2022c). Xtext homepage. https://www.eclipse.org/Xtext/. Last accessed Nov 2022.

Efftinge, S. and Völter, M. (2006). oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32.

Eysholdt, M. and Behrens, H. (2010). Xtext: implement your language faster than the quick and dirty way. In *OOPSLA Companion*, pages 307–309.

Gmys, J., Carneiro, T., Melab, N., Talbi, E.-G., and Tuyttens, D. (2020). A comparative study of high-productivity high-performance programming languages for parallel metaheuristics. *Swarm and Evolutionary Computation*, 57:100720.

Iung, A., Carbonell, J., Marchezan, L., Rodrigues, E., Bernardino, M., Basso, F. P., and Medeiros, B. (2020). Systematic mapping study on domain-specific language development tools. *ESEM*, 25(5):4205–4249.

Kosar, T., Bohra, S., and Mernik, M. (2016). Domain-specific languages: A systematic mapping study. *IST*, 71:77–91.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344.

Neubauer, P., Bergmayr, A., Mayerhofer, T., Troya, J., and Wimmer, M. (2015). Xmltext: from xml schema to xtext. In *SLE*, pages 71–76.

Nordmann, A., Hochgeschwender, N., and Wrede, S. (2014). A survey on domain-specific languages in robotics. In *SIMPAR*, pages 195–206. Springer.

Python Community (2022). Python homepage. https://www.python.org/. Last accessed Nov 2022.

Rodchenkov, M. (2019). Xenia metamodel. https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/model/generated/Xenia.ecore, Last accessed Nov 2022.

Rodchenkov, M. (2020). Xenia homepage. https://github.com/rodchenk/xenia/, Last accessed Nov 2022.

Sredojević, D., Okanović, D., Vidaković, M., Mitrović, D., and Ivanović, M. (2015). Domain specific agent-oriented programming language based on the xtext framework. In *ICIST*, pages 8–11.

Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: eclipse modeling framework*. Pearson Education.