33rd European Modeling & Simulation Symposium
18th International Multidisciplinary Modeling & Simulation Multiconference

ISSN 2724-0029 ISBN 978-88-85741-57-7 © 2021 The Authors.
doi: 10.46354/i3m.2021.emss.010

A Conversion Framework of the Continuous Modeling Languages Based on ANTLR4

Zhen Chen[1] , Lin Zhang[1,*] , Xiaohang Wang[1] , Pengfei Gu[1] and Fei Ye[1]

[1]Beihang University, 37-Xueyuan Road-Haidian, Beijing, 100191, China

[*]Corresponding author. Email address: johnlin9999@163.com

Abstract

Based on the needs of production and life, the modeling and simulation of the continuous system have a very wide range of requirements and applications. Various continuous modeling languages play an important role in the modeling and simulation of such systems. However, the same models built in different languages have to be rebuilt each time, which causes the problem of poor reusability of models between different languages. This paper proposes a conversion framework of the continuous system modeling language based on ANTLR4. And the Modelica to X language conversion experiment using this framework is implemented, whose results achieve high accuracy in syntax check. This framework indicates the method to complete the conversion between different modeling languages so that the same model can be reloaded between different modeling languages, which prevents modeling and simulation personnel from repeatedly modeling the same model, and this makes it easier for the new modeling and simulation language to build a model library.

Keywords: Modeling language conversion; ANTLR4; framework

1. Introduction

The source-to-source conversion between languages refers to the conversion between two programming languages at roughly the same level of abstraction. The converted code structure can be similar to the source code, which can also be significantly changed, but the functions implemented by the code need to remain the same.

Continuous modeling and simulation language is an important branch of a simulation language. This type of language mainly carries on the equation-oriented abstraction to the model object. Among the various system models currently established, the continuous system model is a common portion. There are a large number of continuous models in various fields such as missile vehicle simulation, heat transfer and flow analysis, dynamic analysis, biological and medical model establishment, etc. Therefore, the continuous modeling language plays a huge role in the simulation of continuous systems described by linear and non-linear differential equations.

Since the simulation council formulated the continuous system simulation language CSSL specification, continuous modeling languages or modeling simulation languages that can describe continuous systems have emerged in an endless stream, such as ACSL, DARE-P, DESIR, AnyLogic, Dynamo, etc. The increasingly popular Modelica language for object-oriented and equation-oriented also has perfect support for continuous systems. Although the grammatical details are different, their basic principles are the same. Due to the differences of languages, the models established by different languages can only be reused within the language, which can not be reused among languages. Therefore, if the conversion between different modeling and simulation languages can be realized, some established models can be reused. Besides, it also contributes to the expansion of the language model library. As the foregoing analysis shows, continuous modeling languages follow the same principles and are similar at the abstract level, which provides the possibility for this conversion.

Based on the above analysis, combined with the relevant knowledge of language conversion, this paper proposes a continuous modeling language conversion framework based on ANTLR4, which provides a feasible method for the conversion between different near-source continuous modeling languages.

Section 1 and section 2 of this paper briefly analyze the requirements of modeling language conversion and introduce the related work of language conversion. Section 3 introduces the composition and characteristics of common continuous modeling languages and describes the conversion front-end tool. Section 4 is the key part of this article, which proposes a detailed ANTLR4-based continuous modeling language conversion framework and describes the specific steps of converter construction. Finally, the conversion from Modelica language to X language was carried out, which proved the feasibility of the conversion framework.

2. Related works

For the conversion of programming language source code to source code, Terekhov and Verhoef (2000) state the three traditional mainstream methods of source code to source code conversion: grammar-guided converter, rule-based converter, and model-driven converter.

A grammar-guided converter is similar to a parser, but in the process of parsing, action statements that can produce output are embedded, and intermediate structures without conversion are produced. Grammar-guided conversion is applied in the automatic construction of assembly language (Jinghe Wen, 2005). But the mapping relationship in the text is simple, and this method is difficult to handle complex conversion tasks. The rule-based converter employs a specific mapping relationship and the grammar rules of the input language to achieve translation: the translation engine parses the language according to the grammar of the input language to generate a parse tree and then obtains the output according to the mapping rules. Grammar rules can be completed with the help of a specific rule engine. Rule-based converters are particularly suitable for converting legacy code because the code after conversion should remain similar in structure to that before conversion. In addition, rule-based converters are one of the mainstream methods of early natural language translation methods(Terekhov & Verhoef, 2000). However, the conversion and output of this method are mixed, and it is difficult to flexibly adapt to the output text format. The model-driven converter is the most widely used conversion framework in the industry: input text goes through three steps: parsing and constructing IR, semantic analysis and data structure construction, and generator outputting text. The model-driven converter achieved good effectiveness by subdividing the conversion work, but on the contrary, it also increased the workload. Model-driven converters have achieved good applications in a variety of code application scenarios. For example, Dixun Zhang (2017) takes advantage of the idea of the model-driven converter, employs the Clang compiler, and completes the code conversion from SIMC to SIMD by using HASI and LASI, two intermediate representations. Although this method can achieve better effect, there are many intermediate representations experienced, and the conversion code is complicated to implement. The above three conversion methods only depend on the grammatical rules of the source language, the mapping relationship between input and output, and the grammatical rules of the target language, but have nothing to do with the corpus of the language itself. In addition, once the conversion mechanism adopting the above methods is established, a higher conversion accuracy rate will generally be obtained.

Regarding the conversion of programming languages, in recent years, some researchers have made new attempts based on programming language corpora. For example, the idea of natural language machine translation is applied to the conversion of programming languages. A Phrase-Based Statistical Machine Translation model was trained using parallel databases of two corpora and applied in the converter from Java to C#，and achieved good results(Nguyen et al, 2013). Alon (2019) utilizes the above model to attempt to match the two-way matching between source code and pseudocode. Some researchers utilized seq2seq, a neural network model commonly used in natural language processing, for programming language conversion, and found that the generated function cannot be guaranteed to be compilable or even grammatically correct. So relevant researchers put additional restrictions on the decoder to try to improve this method(Amodio et al, 2017). Feng(2020) presents CodeBERT: a pre-trained model for programming and Natural Languages. Based on Feng, Lachaux(2020) proposed an unsupervised conversion mechanism for programming languages based on deep learning, which achieved good results in the conversion between C++, Java, and Python3, and they developed the conversion system named TransCoder. Once these new attempts are successful, a breakthrough will be made in the scope of language conversion, because these new methods themselves do not require prior knowledge of the language itself. In theory, for any Turing-complete two programming languages, if their corpus is sufficient, they can be converted. However, if the support of the corpus is lacking, it will be difficult to realize the training of the model, and the conversion effect depends to a large extent on the quality of the corpus applied to train the model.

3. Background

3.1. Continuous modeling language

Continuous modeling language is a kind of simulation language for continuous system modeling and simulation, and it is a subset of high-level programming language.

There are two main types of continuous system modeling languages: block diagram-oriented modeling languages and equation-oriented modeling languages. Block diagram-oriented modeling language is the main method for modeling and simulation of continuous systems in the early days. Nowadays, equation-oriented modeling ideas have been adopted by most continuous system modeling languages and become the main feature of this type of modeling and simulation languages. The continuous modeling language is suitable for modeling linear systems or systems described by ordinary differential equations or partial differential equations, and can recognite, analyze and optimize the system(Cellier & Greifeneder, 2013). The analysis part includes predicting the system output when the system structure and external input are given. Recognition can find the "singularity" of the system. Optimization refers to adjusting the system structure or optimizing the system according to certain optimization criteria.

3.2. ANTLR4

ANTLR4 (ANother Tool for Language Recognition) is a powerful compiler front-end tool based on ANTLRMorph6 rule engine, which provides a great traversal for the processing of text or binary files(Parr, 2013). Therefore, this tool is widely applied in the generation of languages, tools, and frameworks. In terms of text language processing, ANTLR4 can generate a grammar parse tree under the action of the rule engine according to the rules of the input grammar, and assist in the generation of traversal tool templates.

ANTLR4's analysis of input text is divided into two major steps, lexical analysis and grammatical analysis. In lexical analysis, it adopts a left-recursive method, and in grammatical analysis, it adopts a recursive descent analysis method based on grammatical rules. For lexical and grammatical analysis, the combined utilization of the lexical analyzer LEX and the grammatical analyzer YACC is also one of the common methods used by programmers to make language applications. LEX clusters the input character stream based on regular expressions, and YACC employs a table-driven parser to parse lexical units. Compared with similar tools, ANTLR4 has the following outstanding advantages:

1. Parser of different kinds of programming languages can be generated, such as Java, Python, C, C++, C#, etc.
2. The generated syntax parse tree can be directly printed out by calling built-in functions, and the code functions of the parser are named by rules, which is obviously user-friendly in debugging.
3. The adopted rule-based recursive descent parser has higher parsing speed.

In addition, the new technology adopted in ANTLR4-adaptive ALL enables the generator to perform analysis on the grammar in a dynamic manner at runtime. The advantage of this method is that it can quickly find the part of the input text that does not conform to the grammar, and avoid ambiguous warnings in previous versions or YACC.

An important function of ANTLR4 is to provide convenience for tree traversal. It provides programmers with two ideas. One is to convert the event triggered when traversing the tree with the walker class into the call of the listener, which is an implicit access mode; The other is to employ the visitors to explicitly visit the child nodes, concentrate the operation of the node itself and the child nodes in the visitor, and conduct node-by-node visits in a top-down manner.

For the visitor, the root node of the grammatical parse tree generated by ANTLR4 will contain the method to visit the node, for example, visitRoot(). The visitor will call the above method when visiting the node. The visitRoot() will call the visit method and add the the child node of the node is passed to visit as a parameter to continue the visit traversal. This completes the traversal process from top to bottom. Figure 1 shows The correspondence between visitors and nodes.

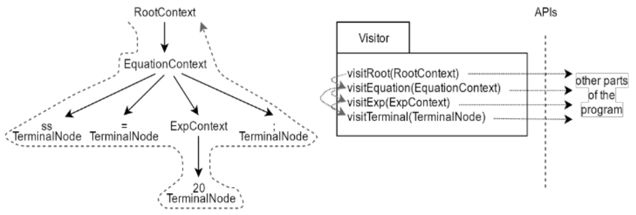


Figure 1. The correspondence between visitors and nodes

4. Conversion framework based on ANTLR4

4.1. Overview

This paper attempts to propose a method that can complete the code conversion between two continuous modeling languages. For the several practices in section 2, the traditional method is mainly based on grammatical rules, while the recent exploration method has higher requirements for the corpus, which requires an abundant bilateral corpus or even a parallel corpus.

It is a challenging task to find the language corpus on both sides of the converter that meets the conditions. However, as a language, modeling language is complete in terms of grammar, and at the beginning of language formulation, language grammar specifications are given. Therefore, in order to complete the general method of code conversion between the two modeling languages, traditional conversion methods can be referred to.

Based on the above analysis, this article proposes a modelling language conversion framework based on ANTLR4, and verified between the two languages. The structure of the framework is shown in the Figure2.

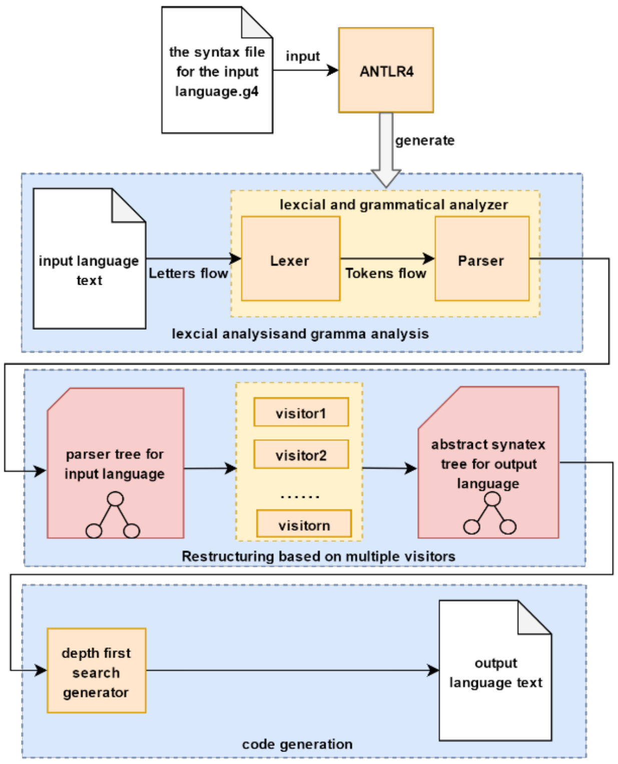

the syntax file for the input language.g4
input
ANTLR4
generate
lexcial and grammatical analyzer
input language text
Letters flow
Lexer
Tokens flow
Parser
lexcial analysisand gramma analysis
parser tree for input language
visitor1
visitor2
......
visitorn
abstract synatex tree for output language
Restructuring based on multiple visitors
depth first search generator
output language text
code generation

Figure 2. Conversion framework based on ANTLR4

The general conversion steps are as follows:

1. Lexer, parser and concrete syntax tree generation. According to the language specification of the input language, the grammar g4 file of the input language can be written. Utilizing the ANTLR4 tool, the lexer and parser are generated, and parse the input text to obtain a concrete syntax tree.
2. Reconstruction and conversion based on distributed multiple visitors. The task of transforming the CST of the input language to the AST of the output language is disassembled, and are assigned to independent visitors. In this step, an abstract syntax tree of the output language that stores the conversion structure needs to be constructed.
3. The text output in depth-first traversal mode. At the tree level, depth-first traversal is performed on the target language abstract syntax tree; at the specific node level, the pre-order, post-order, or middle-order traversal mode is selected based on the node type.

Next, according to the framework, it is divided into three parts to conduct a detailed design analysis for each step.

4.2. Lexical and grammatical analysis

In this part, a lexer and parser are generated for the specific programming language utilizing ANTLR4 and the grammar rules of the input language, which contribute to convert the input text into a concrete syntax tree. The rule file of g4 format input to ANTLR4 needs to be written according to the grammar rules of ANTLR4 grammar and input language. Most modeling languages have relatively complete language specifications at the beginning of language design. After finishing editing the rule file, you can choose the command line method or install the ANTLR4 plug-in in the integrated development environment to generate the lexical parser. The code type of the generated analyzer file can be set by programmer so that the programmer can complete the language processing with a more familiar code, which is also one of the strengths of ANTLR4. Figure 3 shows the flow diagram of lexical and grammatical analysis.

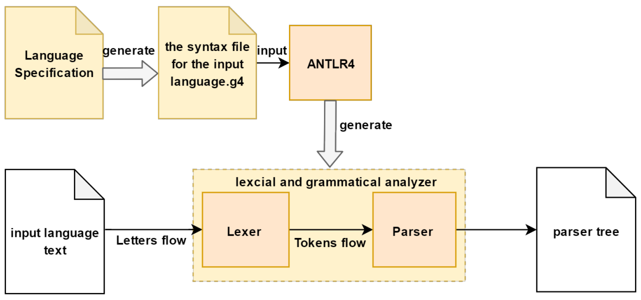


Figure 3. Flow diagram of lexical and grammatical analysis

4.3. Reconstruction and mapping based on multiple visitors

According to the previous introduction, ANTLR4 provides two powerful tools for traversal and operation of the parse tree: listeners and visitors. Compared with a listener, a visitor is more convenient to perform overall operations on a certain node. For some language conversions that are more similar or do not need to be detailed to the leaf nodes, visitors can provide greater convenience. The conversion between modeling languages generally has similarities, especially continuous modeling languages. The description of the system is mostly based on equations, and these equations can sometimes not be disassembled during mapping. Therefore, the framework proposed in this paper employs multiple visitors for mapping operations. Figure 4 shows the flow diagram of mapping based on multiple visitors.

This part needs to complete two tasks. On the one hand, multiple independent visitor arrays need to be constructed to complete various operations on the parse tree, and on the other hand, the abstract syntax tree of the target language needs to be constructed. In order to complete the operation of the concrete syntax tree, each node class of the syntax analysis tree generated by the ANTLR4 tool has a visitor interface. The advantage of this interface is that if the programmer needs to visit the child node through the parent node method in the visitor, the visitor function under the corresponding child node class in the parse tree will be called, and the function returns the corresponding child node in the visitor method. Thus, the visitor method of the child node can be defined in the visitor to operate the child node. So the programmer completes the traversal and operation of the tree outside the analysis tree.

In order to complete the conversion task, according to the principle of compilation, the symbol table of each variable needs to be generated first. Symbol table generation visitor can accomplish this task. The variables in the text are obtained in the order of access under self-direction. If the variable type is referenced from other texts, it can be obtained in other texts after obtaining the reference path. However, the disadvantage of this method is that it requires a lexical and grammatical analysis and traversal of all cited files. After generating the symbol table, the mapping-visitor is created combined with the correspondence between two languages. Moreover, these two visitors are indispensable, and programmers are required to implement additional visitors with corresponding functions according to the needs of specific conversion tasks. The advantage of using distributed multiple visitors is that each visitor can independently implement a relatively single function, which is easy to implement and has strong readability.

Abstract syntax tree is a relatively concise tree structure for describing text. The concrete syntax tree generated by ANTLR4 has a large number of built-in rules, and these rules are only useful when analyzing the input text, but not meaningful for the conversion itself, so it needs to be reduced to a more streamlined abstract syntax tree structure. In addition, constructing the abstract syntax tree of the target language can also facilitate the formation of the output text. The specific details will be described in the next section. The construction of abstract syntax tree follows the following principles:

1. Appropriate declarations need to be constructed according to the target language structure;
2. The order of sentences should be explicitly shown in the design of the node;
3. The AST needs to be flexible enough to quickly add an unknown number of subcodes.

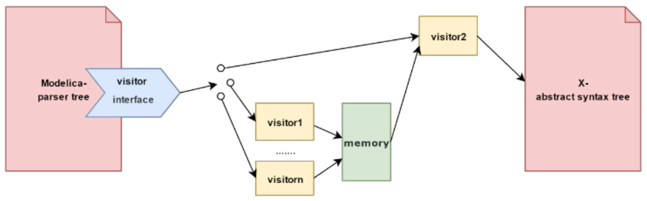
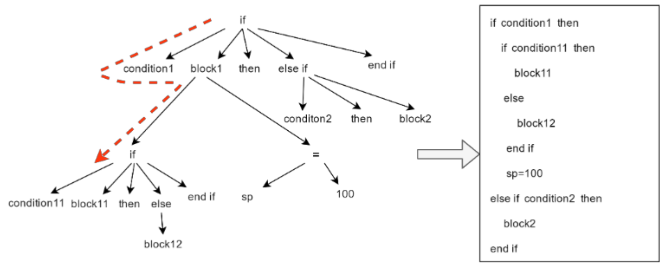


Figure 4. Flow diagram of mapping based on multiple visitors

4.4. The generation algorithm based on depth first search

Different modeling languages have differences not only in describing modeling, but also in text organization. Therefore, in order to easily generate a code format that meets the target language, a more appropriate approach is to perform the parsing conversion and output generated code step by step. After obtaining the concrete syntax tree of the input text, instead of directly outputting it into the target language by parsing through each visitor, it firstly employs the abstract syntax tree of the target language as an intermediate carrier to create a nested output model. By this way, the tree-to-tree mapping is realized, which makes the difficulty of mapping appropriately reduced, and the structure of the target abstract syntax tree can be dynamically adjusted to meet the mapping requirements. In addition, the target language abstract syntax tree constitutes the output nesting model, which can more flexibly meet the organizational requirements of the output language code such as indentation, punctuation, etc. The programmer only need to define the output order for each node of the tree and perform a depth-first traversal of the tree, then the target code can be output. The following Figure 5 is a case of pseudo code and abstract syntax tree.



```
if condition1 then
  if condition11 then
    block11
  else
    block12
  end if
  sp=100
else if condition2 then
  block2
end if
```

Figure 5. The correspondence between Traversal and output

In the case shown in the figure above, there are two main strategies to consider when outputting: depth-first traversal and proper traversal order for nodes. For the entire tree, the depth-first traversal method can output the target text with the correct structure, that is, when a node is traversed, if its child nodes are not empty, the traversal direction is carried out along its child nodes until the leaf node is returned. As for the traversal sequence of a certain node, it needs to be considered in conjunction with the specific node type. For example, in the correspondence between the abstract syntax tree and the code block in the above figure, the traversal output order of the nodes of the if conditional sentence should be preorder traversal, that is, the root node if is output first. The equation "sp=100", due to " =" is the root node, in the actual equation, the equal sign is in the middle, so the output needs to be traversed in the middle order.

5. Experiment

5.1. Languages applied for the conversion

Modelica language is a prevailing multi-domain physical system modeling language. It applies object-oriented and declarative equation modeling ideas to describe the physical world, breaking the domain boundaries of physical models(Fritzson, 2014).

At the language level, Modelica has the following two main characteristics: First, the Modelica physical model is organized by "classes", which are the basic structural elements of the modelica language. Classes generally contain member classes, variables, and equations, some classes can contain only a part of them. There are two types of classes, restricted classes and general classes. Special classes have special restrictions on members, for example, the restricted class connector is applied to describe the connection relationship between models, and the record class is used to describe specific data structures. The function class is specifically used to describe functions, and the general class is represented by class. Second, Modelica's component connection mechanism makes the models have strong reusability and flexibility. Each component has a connector class as an interface for external interaction, and the internal equations of the component model only have interface variables and internal variables, which ensures the reusability of the component. The interface is an instantiation of the connector class, including flow variables and potential variables.

X language is a full-process modeling language for complex products. Due to the complexity of the complex product system structure, there are many disciplines involved. As the complexity of the product increases, the difficulty of design, modeling and simulation will also increase greatly. In order to solve the problem of multi-domain and multi-software collaborative design and simulation, X language bases on the whole process of modeling, unifies system-level design and physical-level simulation. X language includes system-level graphics modeling and simulation-level physical modeling. At the system design level, graphical modeling in X language can employ definition diagrams, state diagrams, connection diagrams to design the system. At the simulation level, X language text can describe and simulate the physical model. The graphic level and the text level have a natural built-in correspondence, which guarantees the whole process of system design.

The conversion of Modelica to X language to be done in this section is aimed at the text level of X language, so the following introduction will focus on the text level. At the text modeling and simulation level, X language absorbs the advanced modeling ideas of Modelica language, adopts object-oriented modeling methods and the idea of declarative equation modeling, but in terms of the applicable objects of modeling, X language makes a full range of improvements and supplements, it increases support for the agent model and in view of the shortcomings of Modelica's main application of continuous physical models, innovates the modeling method of discrete systems, and improves the modeling capabilities of discrete systems. Similar to Modelica, the class is its basic element. In addition to general class, the restricted class includes continuous, discrete, couple, agent, record, function, and connector. For a specific class, there are clear members. For example, the continuous class contains two sections of definition and equation, couple contains two sections of definition and connection, and the member sections both have clear declarations. The comparison between the special classes of Modelica and X language is shown in Table 1.

5.2. Conversion performance

The Modelica standard library is a model package maintained by the Modelica Association, which covers basic models in some fields, as well as common physics and computer constants, which facilitates the application of Modelica by scientists and engineers. This experiment intercepts three different classes in the Modelica model library and converts them.

Table 2. Special classes of Modelica and X language.

Modelica	X language
model	continuous
function	discrete
connector	agent
record	couple
type	function
package	connector
	record

5.2.1. Grammar check accuracy and usability

We randomly select 10 samples of each of the three classes showing in the Table 2. Since the conversion framework strictly considers the code structure of the X language when doing the mapping, all conversion structures can pass the syntax check. Among them, auxiliary classes such as connector, function, and model classes without external citations can be directly applied in the simulation application of the X language. Part of the model class models refer to external classes iteratively, so the result of this type of single model text conversion cannot be directly used for X language simulation.

Table 1. Conversion performance about grammar check and application.

Class type	Number of sample models	Number of passing grammar check	Number of applying directly in simulation
function	10	10	10
model	10	10	5
connector	10	10	10

5.2.2. Amount of code before and after conversion

Since the Modelica model library file has been applied in engineering practice, it contains a large number of comment sentences, which are also counted as the number of Modelica text lines, so it needs to be simplified. The following figure reflects the number of lines of original Modelica text code, the number of simplified lines, and the number of lines after conversion.

It is easy to find from the experimental results that the amount of code in the simplified Modelica is basically the same as the X language text, but is quite different from the amount of code in the original Modelica library. From the analysis of the amount of code, the experimental results indicate Modelica and X language are similar in the abstract level.

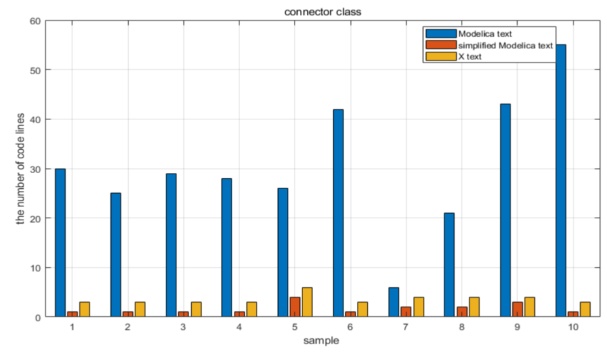


Figure 6. Code amount of connector class

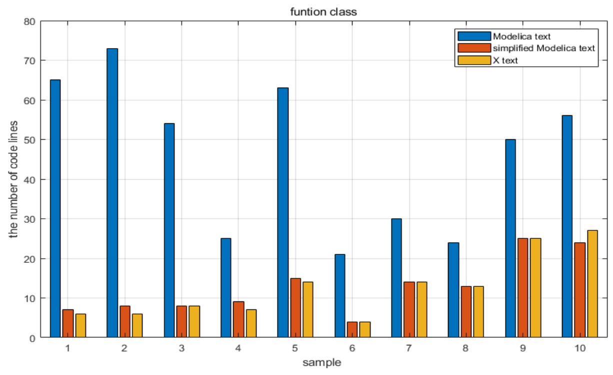


Figure 7. Code amount of function class

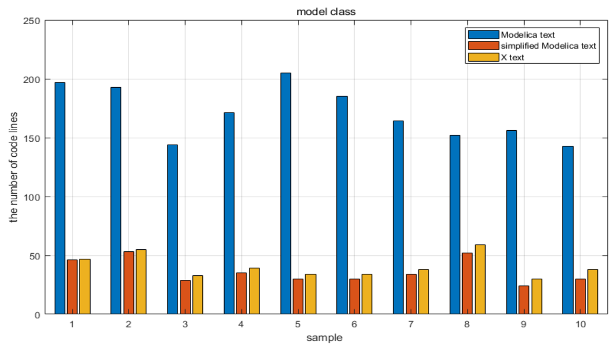


Figure 8. Code amount of model class

5.2.3. Conversion time

The figure below is a statistical comparison of the conversion time of each model. Among them, the model class model in Figure 9 does not reference other classes externally. It can be found that the conversion time of each text in Figure 9 is basically kept below 2s. In Figure 10, the conversion time of the two types of model files with and without external citation classes is compared. We can find that the time spent with externally referenced classes is significantly longer, because it is necessary to search for the source of the reference and parse the file where the referenced class is located. During this period, various data needs to be manipulated, thereby increasing the conversion time.

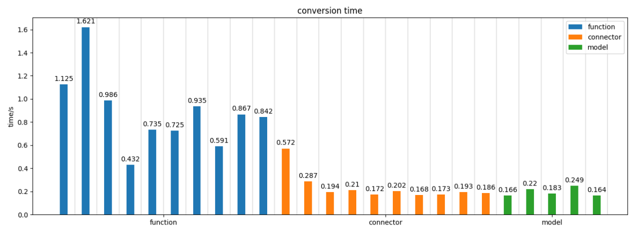


Figure 9. Conversion time among different classes

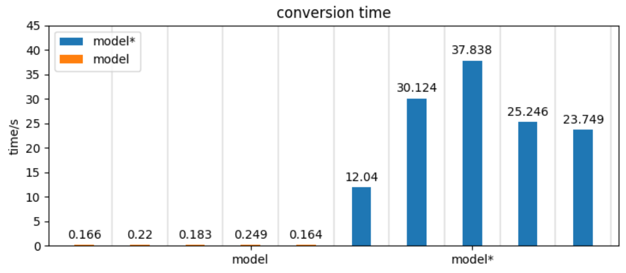


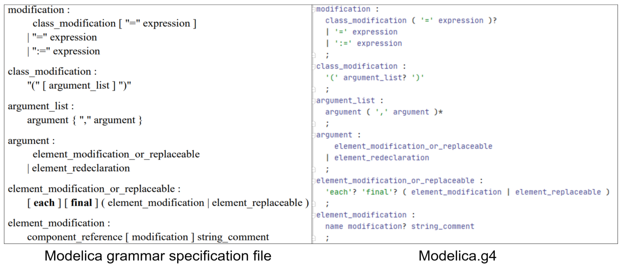
Figure 10. Conversion time between model with and without external citation class

5.2.4. Conversion demonstration

The following is a demonstration of the conversion of a model class file. The input is a model class text named Prismatic, whose index in the Modelica standard library is Machanics. MultiBody.- Joints. Prismatic, which contains multiple member types: external member classes, variables, and equations. The model has a total of 201 lines of code.

Step1: Editing a rule file and generating a parsing tool utilizing ANTLR4. Refer to appendix B.2 of the latest Modelica Language Specification version 3.4 published on Modelica official website(Modelica Specification Version3.4, 2017), the ANTLR4 rule file Modelica.g4 can be edited. By the Modelica.g4 file and ANTLR4, lexer and parser coded in Python3 are generated, whitch parse the input text to obtain a concrete syntax tree.

Figure 11 shows the Modelica grammar specification file and Modelica.g4 file.



```
modification :
      class_modification [ "=" expression ]
    | "=" expression
    | ":=" expression

class_modification :
    "(" [ argument_list ] ")"

argument_list :
    argument { "," argument }

argument :
      element_modification_or_replaceable
    | element_redeclaration

element_modification_or_replaceable :
    [ each ] [ final ] ( element_modification | element_replaceable )

element_modification :
    component_reference [ modification ] string_comment
```

```
modification :
  class_modification ( '=' expression )?
  | '=' expression
  | ':=' expression
  ;
class_modification :
  '(' argument_list? ')'
  ;
argument_list :
  argument ( ',' argument )*
  ;
argument :
    element_modification_or_replaceable
  | element_redeclaration
  ;
element_modification_or_replaceable :
  'each'? 'final'? ( element_modification | element_replaceable )
  ;
element_modification :
  name modification? string_comment
  ;
```

Figure 11. Modelica grammer specification and modelica.g4

Figure 12 shows the input and output of this step, where the parser tree is a visual display of CST.

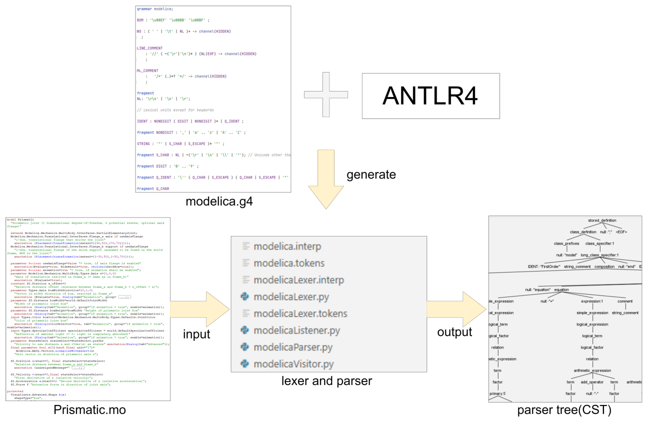
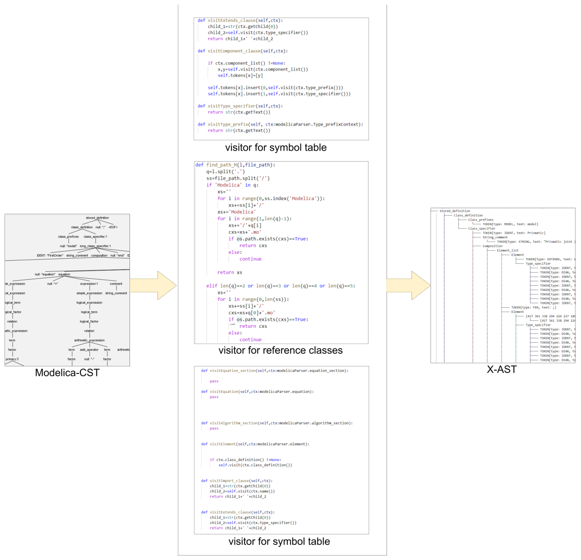


Figure 12. Lexical and grammatical analysis of model Prismatic

Step2: Editing symbol-generation visitor, constructing AST of X language, editing mapping visitors according rules. In this conversion, the model member class contains external citation , variables, and interfaces. And there are connection elements in the equation. In addition, Modelica and X language have different organization of models, and the equation part of the Modelica file contains connection classes, so the source file needs to be disassembled into two files. Therefore, the following steps need to be operated:

1. Obtain the type of external citation class from the cited file.
2. The "connect" subtree in the equation needs to be intercepted and written into the couple class file, and the remaining part should be written into the continue class file.

Figure 13 shows the visitors, input and output of this step.



```
def visitExtends_clause(self,ctx):
    child_1=str(ctx.getChild(0))
    child_2=self.visit(ctx.type_specifier())
    return child_1+' '+child_2

def visitComponent_clause(self,ctx):

    if ctx.component_list() !=None:
        x,y=self.visit(ctx.component_list())
        self.tokens[x]=[y]

    self.tokens[x].insert(0,self.visit(ctx.type_prefix()))
    self.tokens[x].insert(1,self.visit(ctx.type_specifier()))

def visitType_specifier(self,ctx):
    return str(ctx.getText())

def visitType_prefix(self, ctx:modelicaParser.Type_prefixContext):
    return str(ctx.getText())
```

visitor for symbol table

```
def find_path_M(l,file_path):
    q=l.split('.')
    ss=file_path.split('/')
    if 'Modelica' in q:
        xs=''
        for i in range(0,ss.index('Modelica')):
            xs+=ss[i]+'/'
        xs+='Modelica'
        for i in range(1,len(q)-1):
            xs+='/'+q[i]
            cxs=xs+'.mo'
            if os.path.exists(cxs)==True:
                return cxs
            else:
                continue

        return xs

    elif len(q)==2 or len(q)==3 or len(q)==4 or len(q)==5:
        xs=''
        for i in range(0,len(ss)):
            xs+=ss[i]+'/'
            cxs=xs+q[0]+'.mo'
            if os.path.exists(cxs)==True:
                return cxs
            else:
                continue
```

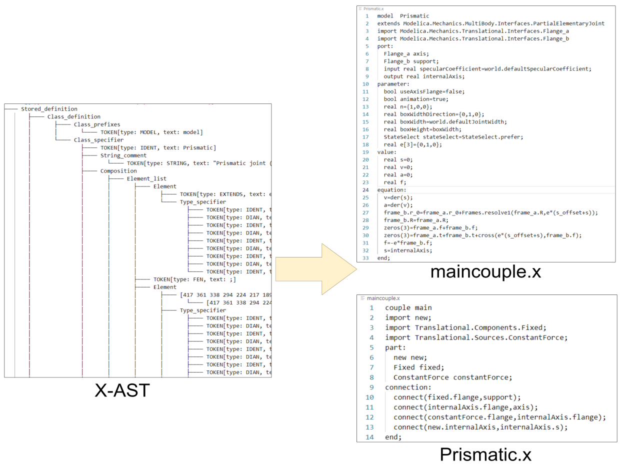
visitor for reference classes

```
def visitEquation_section(self,ctx:modelicaParser.equation_section):

    pass

def visitEquation(self,ctx:modelicaParser.equation):
    pass



def visitAlgorithm_section(self,ctx:modelicaParser.algorithm_section):
    pass


def visitElement(self,ctx:modelicaParser.element):


    if ctx.class_definition() !=None:
        self.visit(ctx.class_definition())


def visitImport_clause(self,ctx):
    child_1=str(ctx.getChild(0))
    child_2=self.visit(ctx.name())
    return child_1+' '+child_2


def visitExtends_clause(self,ctx):
    child_1=str(ctx.getChild(0))
    child_2=self.visit(ctx.type_specifier())
    return child_1+' '+child_2
```

visitor for symbol table

Figure 13. Mapping from Modelica-CST to X-AST

Step3: Depth first traversal is adopted in the traversal of the whole tree, and the output order of each node is determined by the output function in the AST node class. Figure 14 reflects the input and output of this step.



```
├── Stored_definition
    ├── Class_definition
    │   ├── Class_prefixes
    │   │   └── TOKEN[type: MODEL, text: model]
    │   └── Class_specifier
    │       ├── TOKEN[type: IDENT, text: Prismatic]
    │       ├── String_comment
    │       │   └── TOKEN[type: STRING, text: "Prismatic joint (
    │       ├── Composition
    │       │   ├── Element_list
    │       │   │   ├── Element
    │       │   │   │   ├── TOKEN[type: EXTENDS, text: e
    │       │   │   │   └── Type_specifier
    │       │   │   │       ├── TOKEN[type: IDENT, t
    │       │   │   │       ├── TOKEN[type: DIAN, te
    │       │   │   │       ├── TOKEN[type: IDENT, t
    │       │   │   │       ├── TOKEN[type: DIAN, te
    │       │   │   │       ├── TOKEN[type: IDENT, t
    │       │   │   │       ├── TOKEN[type: DIAN, te
    │       │   │   │       ├── TOKEN[type: IDENT, t
    │       │   │   │       ├── TOKEN[type: DIAN, te
    │       │   │   │       └── TOKEN[type: IDENT, t
    │       │   │   ├── TOKEN[type: FEN, text: ;]
    │       │   │   ├── Element
    │       │   │   │   ├── [417 361 338 294 224 217 189
    │       │   │   │   │   └── [417 361 338 294 224
    │       │   │   │   ├── Type_specifier
    │       │   │   │   │   ├── TOKEN[type: IDENT, t
    │       │   │   │   │   ├── TOKEN[type: DIAN, te
    │       │   │   │   │   ├── TOKEN[type: IDENT, t
    │       │   │   │   │   ├── TOKEN[type: DIAN, te
    │       │   │   │   │   ├── TOKEN[type: IDENT, t
    │       │   │   │   │   ├── TOKEN[type: DIAN, te
    │       │   │   │   │   ├── TOKEN[type: IDENT, t
    │       │   │   │   │   ├── TOKEN[type: DIAN, te
```

X-AST

Prismatic.x

```
model  Prismatic
extends Modelica.Mechanics.MultiBody.Interfaces.PartialElementaryJoint
import Modelica.Mechanics.Translational.Interfaces.Flange_a
import Modelica.Mechanics.Translational.Interfaces.Flange_b
port:
  Flange_a axis;
  Flange_b support;
  input real specularCoefficient=world.defaultSpecularCoefficient;
  output real internalAxis;
parameter:
  bool useAxisFlange=false;
  bool animation=true;
  real n={1,0,0};
  real boxWidthDirection={0,1,0};
  real boxWidth=world.defaultJointWidth;
  real boxHeight=boxWidth;
  StateSelect stateSelect=StateSelect.prefer;
  real e[3]={0,1,0};
value:
  real s=0;
  real v=0;
  real a=0;
  real f;
equation:
  v=der(s);
  a=der(v);
  frame_b.r_0=frame_a.r_0+Frames.resolve1(frame_a.R,e*(s_offset+s));
  frame_b.R=frame_a.R;
  zeros(3)=frame_a.f+frame_b.f;
  zeros(3)=frame_a.t+frame_b.t+cross(e*(s_offset+s),frame_b.f);
  f=-e*frame_b.f;
  s=internalAxis;
end;
```

maincouple.x

maincouple.x

```
couple main
import new;
import Translational.Components.Fixed;
import Translational.Sources.ConstantForce;
part:
  new new;
  Fixed fixed;
  ConstantForce constantForce;
connection:
  connect(fixed.flange,support);
  connect(internalAxis.flange,axis);
  connect(constantForce.flange,internalAxis.flange);
  connect(new.internalAxis,internalAxis.s);
end;
```

Prismatic.x

Figure 14. The code generation from AST

6. Conclusion

In this article, we show a continuous modeling language conversion framework based on ANTLR4 and apply this framework to the experiment of conversion between two languages. The experimental results show the feasibility of the framework.

However, this article only studies the conversion of continuous models from Modelica to X language and does not explain the conversion of discrete models, which needs to be supplemented in later work.

In the future, more conversions between other languages are needed to verify its generality as a framework. In addition, visitors need to be designed and classified in more detail, forming a template to facilitate the use of developers.

Funding

This work is supported by the National Key R&D Program of China (GrantNo.2018YFB1701602).

References

Alon, U. , R Sadaka, Levy, O. , & Yahav, E. . (2019). Structural language models of code.

Amodio, M. , Chaudhuri, S. , & Reps, T. . (2017). Neural Attribute Machines for Program Generation.

Cellier, F. E. , & Elmqvist, H. . (1993). Automated formula manipulation supports object-oriented continuous-system modeling. Control Systems IEEE, 13(2), 28-38.

Parr. The definitive antlr4 reference (2013)| forum - heise online. Heise Zeitschriften Verlag.

Dixun Zhang. (2017). Code conversion technology from domain programming language SIMC to SIMD (Master's thesis, Jilin University).

Feng, Z. , Guo, D. , Tang, D. , Duan, N. , Feng, X. , & Gong, M. , et al. (2020). Codebert: a pre-trained model for programming and natural languages.

Lachaux, M. A., Roziere, B., Chanussot, L., & Lample, G. (2020). Unsupervised translation of programming languages. arXiv preprint arXiv:2006.03511.

Fritzson, P. (2014). Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach. John Wiley & Sons.

Jinghe Wen. (2005). The application of grammar-guided translation in automatic assembly program construction (Doctoral dissertation).

Nguyen, A. T., Nguyen, T. T., & Nguyen, T. N. (2013, August). Lexical statistical machine translation for language migration. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (pp. 651-654).

Terekhov, A. A. , & Verhoef, C. . (2000). The realities of language conversions. IEEE Software, 17(6), p.111-124.

Modelica Association. (2017).Modelica Specification Version3.4. Retrieved from https://www.modelica.org/association