Lightweight Software Language Processing Using Antlr and CGTL

Kevin Lano[a] and Qiaomu Xue
King's College London, London, U.K.

Keywords: MDE, MDE Usability, Language Engineering.

Abstract: Software complexity has become a significant social problem, which MDE endeavours to alleviate, however MDE approaches and tools often introduce additional complexity which prevents general software practitioners from benefiting from MDE solutions. In this paper we present an alternative approach for MDE in the domain of language processing, using lightweight tools (Antlr and CGTL) suitable for general industrial use. We evaluate the approach on tasks of DSL definition, software abstraction, and program translation, based on our experience with industrial applications of MDE.

1 INTRODUCTION

The agility and flexibility of MDE tools and methods has been identified as a key issue in improving the uptake of MDE by industry (Abrahao et al., 2017; Bucchiarone et al., 2020; Whittle et al., 2017), however the principal emphasis of MDE tools has been on providing powerful functionality, and they often depend upon substantial technology stacks and have complex GUIs. They have been orientated towards use by researchers, rather than by practitioners (Clark and Muller, 2012). In some cases there has been no formal usability testing of the tools. From an industry perspective, such tools can involve significant risk, because they require a high degree of specialised expertise to use, and their long-term support may be uncertain. In our experience of working with MDE adopters in the finance industry, this sustainability issue has been a key consideration which led companies to prefer simpler tooling which would be usable by general software practitioners.

Software language processing tasks are essential activities in MDE, and include: (i) definition of specialised domain-specific languages (DSLs) together with supporting tools; (ii) abstraction of software in 3GLs to models; (iii) code generation of programming language code from models; (iv) translation from one software language to another.

From experience with industrial cases of DSL definition and program translation we have evolved a pragmatic language processing approach which is focussed on software language grammars and the processing of parse trees from these grammars. An established tool for building grammars and parsers is Antlr (Antlr, 2022a). This is a lightweight tool usable by general software practitioners. There are Antlr grammars for over 230 source languages, including all the main 3GLs[1]. To process the parse trees produced by Antlr, we use a special-purpose DSL, termed *Concrete Grammar Transformation Language* (CGTL). This is a text-to-text transformation language based on the CSTL code generator language of (Lano and Xue, 2020), but has been generalised to process the parse trees of any software language. CGTL is based on concepts of language grammar and can be used by developers without high software modelling expertise. We consider two principal use cases for the combination of Antlr and CGTL: (i) the definition of textual DSLs and tools to produce documentation or code from DSL models; (ii) translation of programs from one 3GL to another. Both tasks involve translating one software language $\mathcal{L}_1$ to another, $\mathcal{L}_2$, by successive steps of parsing and processing (Figure 1).

We also investigate the application of the approach for performing model transformations such as refactorings.

We address the following research questions:

RQ1. Can the Antlr and CGTL software language processing approach be effectively used for DSL definition and tooling?

RQ2. Can the approach be effectively applied to support program translation between 3GLs?

Section 2 describes our experience of the barriers to introducing MDE in industry. Section 3 introduces CGTL, Section 4 describes the use of Antlr

[a] https://orcid.org/0000-0002-9706-1410

[1] https://github.com/antlr/grammars-v4

Lano, K. and Xue, Q.
Lightweight Software Language Processing Using Antlr and CGTL.
DOI: 10.5220/0011623000003402
In *Proceedings of the 11th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2023)*, pages 19-30
ISBN: 978-989-758-633-0; ISSN: 2184-4348

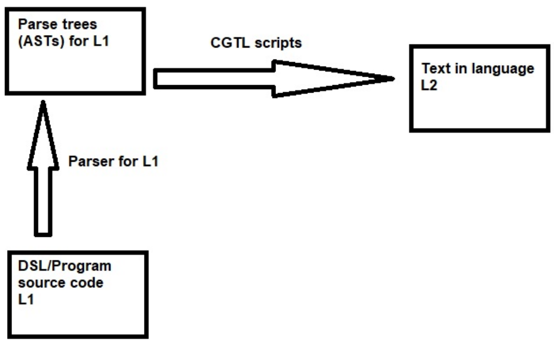

Figure 1: Antlr/CGTL language processing process.

and CGTL for DSL definition and tooling, and Sections 5, 6 describe the use of Antlr and CGTL for program abstraction and translation. Section 7 discusses the use of Antlr/CGTL for general model transformation. Section 8 gives an evaluation, Section 9 compares our approach with other related work, Section 10 discusses limitations and future work, Section 11 considers threats to validity, and Section 12 gives conclusions. The Appendix gives details of CGTL semantics.

2 BARRIERS TO MDE ADOPTION

Between 2019 and 2022 we advised a major financial services corporation on the adoption of MDE to improve their software development and maintenance processes. There appeared to be a strong business case to support the adoption, in terms of rationalisation of assets and reduced time-to-market of products, however there were substantial barriers: (i) lack of experience with MDE and lack of MDE skills; (ii) the need for long-term support of MDE tooling; (iii) disruption of existing practices and organisational relationships.

To address these barriers, an approach was evolved which used lightweight tooling (principally Antlr) and simple DSLs, which could be used by general software practitioners without advanced MDE skills. The process was essentially the same as in Figure 1. Based on this experience, we formalised the CGTL language, and identified how this could be used to support multiple MDE language processing tasks.

3 CGTL AND CSTL

CSTL was created in order to provide a rapid means of writing code generators from UML/OCL specifications to 3GLs (Lano and Xue, 2020). Our experience with writing large code generators in Java, OCL (Lano et al., 2017) and EGL motivated the definition of a simple and concise language to express code production from models. Some example CSTL rules for mapping binary expressions to Java are:

```
BinaryExpression::
_1 & _2 |-->_1 && _2
_1 or _2 |-->_1 || _2
```

Execution is based on pattern matching of source elements with the LHS of rules, and text substitution of mapped target elements into the RHS of rules.

CGTL retains the same syntax and concepts as CSTL, but instead of operating on UML/OCL model data, it processes the parse trees or abstract syntax trees (ASTs) of software languages – in principle for any source language that has a grammar. The metamodel of AST terms which we use to represent parse trees is shown in Figure 2. The *features* map of *ASTTerm* records information about the language elements represented by terms, such as their type. This information can be set by CGTL actions and read by CGTL conditions. Composite terms with tag *tg* and *n* subterms are written as (*tg* t_1 ... t_n). Basic terms with tag *tg* and value *v* are represented as (*tg v*).

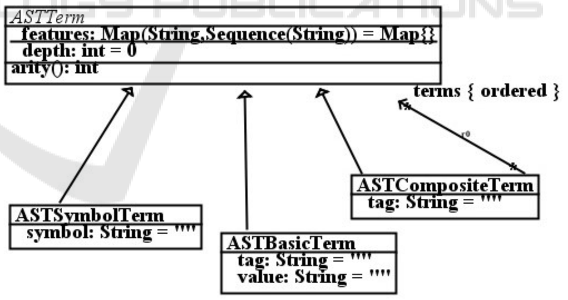

Figure 2: Metamodel of parse trees.

The style of CGTL rules is closely related to the form of standard BNF grammar productions. For example, a CGTL rule to process the ASTs produced by an Antlr grammar rule

```
parameterDecln:
  type identifier;
```

would have a LHS consisting of two metavariables _1, _2 corresponding to the declaration type and identifier parts. A complete CGTL rule could be:

```
_1 _2 |--> _2 : _1
```

This would transform a declaration `int x` into `x : int` if there were no rules for types or identifiers. The rule would be in a ruleset for *parameterDecln*:

```
parameterDecln::
_1 _2 |--> _2 : _1
```

Other alternative productions for *parameterDecln* would be processed by other rules in the same ruleset. For example, there could be a further production to define default parameter values:

```
parameterDecln:
  type identifier ('=' expression)?;
```

This extended definition would be handled by two rules:

```
parameterDecln::
_1 _2 = _3 |--> _2 : _1 := _3
_1 _2 |--> _2 : _1
```

More details of CGTL syntax and semantics are given in the Appendix.

CGTL incorporates several extensions to CSTL: (i) pattern-matching of a single variable against multiple subchildren of a parse tree; (ii) recursive application of a ruleset; (iii) script parameters; (iv) nested rule/script invocation in rule conditions; (v) dynamic loading of scripts.

These changes, together with rule actions, mean that CGTL is a Turing-complete language, thus it can carry out more powerful language processing than the original CSTL or other assignment-free languages such as Antlr's StringTemplate (Antlr, 2022b).

Dynamic loading means that when a script *g* is invoked in a rule RHS as *_i‘g*, the script *g* is loaded from the *cg* directory, if it is not already in memory.

4 DEFINITION OF DSLs

DSLs have been widely used to specify specialised software functionalities or categories of software applications, for example machine learning systems (Zucker and D‘Leeuwen, 2020) and text processing (Desai et al., 2016). Well-known notations such as regular expressions and SQL can also be regarded as DSLs. Use of a DSL to specify software reduces (in principle) the effort required to analyse models and generate code, compared to the use of more general-purpose notations such as UML and OCL. However, DSLs need careful design to ensure that their constructs are appropriate for the intended domain, and can be used by the intended end users. A DSL often needs to evolve, eg., to add new features or constructs. While DSLs can be represented as metamodels, another effective means for defining a textual DSL is by a language grammar. Tools to process DSL models (expressed as texts conforming to the DSL grammar) can operate on the parsed DSL syntax trees produced by a parser. In some cases it is possible to perform processing during parsing by attaching actions to the production rules of the grammar. For example, Antlr provides an action notation to perform processing during parsing, however this approach has limited functionality, and results in the execution order of actions being tied closely to that of parsing.

In order to decouple parsing and processing, we defined the dedicated CGTL language and associated tools to process parse trees of any language, and to produce text in a wide range of formats, including documentation, HTML, XML or programming language code. The structure of the CGTL scripts for a language L will generally follow the structure of the grammar for L. Each grammar category r of L will have a corresponding ruleset r:: in a script for L, to process L ASTs with tag r. Each grammar production p of r should have a corresponding rule of r:: which can process L elements parsed using p. Using the *antlr2cstl* script, given an Antlr grammar file $LParser.g4$ for L, an outline CGTL script can be automatically produced with the appropriate rulesets and rule LHSs to process trees produced by $LParser.g4$[2]. The DSL tool designer then needs to complete the rule definitions to achieve the required transformation.

5 LANGUAGE TRANSLATION

The management and maintenance of software, especially of legacy software, has become a significant social problem (Agarwal et al., 2022) which costs increasing human and financial resources to tackle. Critical software systems exist in antiquated languages such as COBOL or old versions of C, and need to be modernised in order that they can be effectively used and maintained. The costs and time required for manual software modernisation can be prodigious. For example, migration of 35 million lines of a critical banking system written in Python 2.7 to Python 3 took over 3 years of effort at JP Morgan (Sanders, 2019). Automated program translation is therefore an attractive alternative to manual translation or redevelopment of software assets. To support such capabilities we used CGTL to implement abstraction transformations from Java, C, Visual Basic 6 (VB6), Cobol85 and JavaScript programs to UML/OCL.

To translate programs of source language $\mathcal{L}_1$ to

[2]Antlr's grammar is itself defined by an Antlr grammar.

programs of target language $\mathcal{L}_2$, we use an Antlr parser for $\mathcal{L}_1$ to produce parse trees, which are then input to an abstraction transformation for $\mathcal{L}_1$, written using CGTL. The output is a UML/OCL specification in textual form, consisting of class specifications with data features and operations, and use cases defining global processing, such as application initialisation. The specification may utilise the operations of OCL libraries. We have provided additional libraries to represent program semantics for aspects such as files, dates, exceptions and iterators, which are not present in standard OCL (Section 6). Forward engineering using CGTL code generators or other MDE code generation techniques is then employed to map the abstracted specification to the target language $\mathcal{L}_2$.

We use the AgileUML (Eclipse Agile UML project, 2022) and Eclipse OCL (Eclipse, 2022) versions of OCL to represent program semantics. To define program executable behaviour, we adopt the AgileUML textual notation for UML structured activities. This is similar to the extended OCL of (Buttner and Gogolla, 2014), and observes a strict hierarchical relation between expressions and statements, ie., statements cannot occur as subparts of expressions.

5.1 Abstraction of Expressions and Statements

Programming language expressions can be abstracted to OCL expressions using a CGTL script. In many cases there are direct translations from program constructs to corresponding OCL expressions, for example, C and Java array accesses and bitwise negation operator expressions translate directly to OCL equivalents:

```
expression::
_1 [ _2 ] |-->_1->at(_2 + 1)

˜ _1 |-->-(_1 + 1)
```

However in some cases it is necessary to interpret the source language elements using further semantic details, if there is no exact equivalent of the source element in OCL. For example, the *isalpha* and *isdigit* C functions on characters can be interpreted as:

```
isalpha ( _1 ) |-->
  (_1)->byte2char()->isMatch("[a-zA-Z]")
isdigit ( _1 ) |-->
  (_1)->byte2char()->isMatch("[0-9]")
```

The main forms of programming language statements also translate directly into the pseudocode notation of UML activities used in AgileUML, eg., for C we have:

```
selectionStatement::
if ( _1 ) _2 |-->
    if _1\n    then\n  _2\n
    else skip ;\n

if ( _1 ) _2 else _3 |-->
    if _1\n
    then\n  _2\n
    else\n  (_3)\n
```

However, statements with unstructured control flow, such as goto, labelled break/continue, and switch statements, need to be transformed into structured forms via the abstraction process.

5.2 Abstraction of Features and Classes

Application-specific classes defined in a program are mapped to corresponding classes in the UML/OCL representation, with their data features and methods mapped to attributes and operations in UML. C structs, VB6 and Cobol85 records and JavaScript constructor functions are also abstracted as UML classes. Enum definitions are abstracted as enumerated types. The CGTL rules to perform this abstraction for C include:

```
structOrUnionSpecifier::
struct _1 { _2 } |-->class _1\n{_2\n}\n\n
union _1 { _2 } |-->class _1\n{_2\n}\n\n
struct _1 |-->_1
union _1 |-->_1

structDeclarationList::
_* |-->_*
_1 |-->_1

enumSpecifier::
enum _1 { _2 } |-->enumeration _1\n{_2\n}\n\n
enum _1 |-->_1

enumeratorList::
_1 |-->_1
_* |-->_*

enumerator::
_1 = _2 |-->_1
_1 |-->_1

enumerationConstant::
_1 |-->  literal _1;\n
```

These rules preserve the structure of the source program in the abstracted representation, facilitating traceability.

In the case of C, VB6, Cobol85 and JavaScript there may be data and operations with a global scope. These are represented as features of a new class representing the entire program.

6 OCL EXTENSION LIBRARIES

OCL extensions and libraries are provided in order to represent common programming language aspects as follows (Lano et al., 2022):

- First-class function types $Function(S,T)$ are added, together with λ-abstraction expressions *lambda x : T in expr* and application operator $f \rightarrow apply(x)$.
- Library component *OclDate* is added to represent dates and times.
- Facilities for byte processing and number format conversions are added in library component *MathLib*. *MathLib* also supports random number generation and bitwise and/or/xor operations.
- *OclType* is extended to enable inspection of the data and behaviour features of each *OclType* instance.
- Exception handling is represented by *try catch finally* statements and a class *OclException* of exceptions, together with subclasses for more specialised forms of exception, such as *IndexingException*.
- A library class *OclIterator* is defined which supports the creation of iterators for collections. This is also used to represent *generator* functions and database result sets.
- A library class *OclFile* to represent files and streams.
- *OclProcess* supports the creation of threads and OS processes and querying the environment.
- *OclDatasource* models SQL databases, TCP sockets and HTTP connections.

7 MODEL TRANSFORMATIONS

General model transformation (MT) tasks include mapping from models of one language L_1 to models of another, L_2, or refactoring (updating in-place) a model of a single language. The usability of MT languages such as ATL, QVTr and ETL has been questioned (Burgueno et al., 2019b), and the level of industrial adoption of these languages remains low. Effective use of an MT language requires detailed knowledge of the structure of models for the source and target languages involved, ie., knowledge of their metamodels. Model data is typically a graph structure, with intricately interlinked elements. MT rules need to navigate and assemble such links in the correct way.

CGTL could be applied to MT tasks by: (i) producing an AST from a source language model; (ii) processing the AST using CGTL rules; (iii) parsing the text result into the target language using a parser/model assembler which creates a target model from target language text.

The advantage of this scheme is that the CGTL rules can express the key idea of the transformation, without needing to navigate or assemble models. The rules do not depend on the metamodel features. The task of ensuring consistent target model structure is delegated to the model assembler. The disadvantage is the additional processing cost of AST production and target language parsing.

An example of a simple refactoring transformation is expression simplification/normalisation, which can be expressed for OCL expressions by the CGTL rules:

```
OclBasicExpression::
_1 |-->_1

OclBinaryExpression::
0 + _1 |-->_1
_1 + 0 |-->_1
_1 + _1 |-->(_1 * 2)
_1 + _2 |-->(_1 + _2)
_1 - 0 |-->_1
_1 - _1 |-->0
_1 - _2 |-->(_1 - _2)
1 * _1 |-->_1
_1 * 1 |-->_1
_1 * _1 |-->(_1)->sqr()
_1 * _2 |-->(_1 * _2)
```

This transforms an input expression such as $x*x+x*x$ into $((x) \rightarrow sqr()*2)$.

To express this as a model refactoring in a MT language such as ATL, ETL or QVTr would be non-trivial, because the pattern matching and rearrangement of links between elements would need to be handled explicitly. For example, the rule replacing $\_1*\_1$ by $(\_1) \rightarrow sqr()$ could be defined in ATL as:

```
rule ReplaceMultBySqr {
from be : IN!OclBinaryExpression (
    be.operator = '*' and
    be.left.toString() = be.right.toString())
to be1 : OUT!OclUnaryExpression (
  operator <- '->sqr',
  argument <- be.left,
  argument.hasBracket <- true,
  container <- be.container
) }
```

Knowledge of the exact names, types and multiplicities of metamodel features is necessary in order to write such rules. Indeed this tight dependence of transformation rules on the metamodel could lead to high maintenance costs if the metamodel changes.

In terms of expressiveness, CGTL is only able to inspect local data of one AST and its subparts, and cannot search global data, such as all instances of a metaclass. Thus model transformation rules which depend upon two or more input parameters, or which need to refer to $E.allInstances()$ for a metaclass E could not be expressed as CGTL rules.

8 EVALUATION

In this section we evaluate our approach with respect to the research questions of Section 1. All artefacts used in this evaluation are provided at zenodo.org/record/7414171.

8.1 RQ1: DSL Definition and Support

We evaluate the approach by defining two DSLs: (i) for natural language processing (NLP) in requirements formalisation (RF); (ii) for mobile app specification.

8.1.1 Natural Language Processing for Requirements Formalisation

NLP is a key technique for many applications which process or produce natural language text or speech. In particular, NLP has been used for the formalisation of software requirements expressed in natural language documents (Burgueno et al., 2021; Zaki-Ismail et al., 2022; Zhao et al., 2020). Based on a systematic survey of NLP research in this area, we identified processing steps, actions and data which are widely used in NLP for RF, and codified these as a simple DSL with a syntax based on SQL.

NLP activities operate on datasets of elements such as texts or ASTs, and can be categorised as:

- Dataset loading and saving, or checkpoint creation and saving during a protracted process.
- Applying a transformation to each dataset element to produce a new dataset, eg., to tokenise raw text elements into sequences of sentences or words, or to apply part-of-speech (POS) tagging to words.
- Filtering a dataset to remove elements that fail to satisfy a criterion, eg., to remove items that do not have a correct grammatical structure to be valid sentences.
- Analysing the accuracy of some classification procedure on a dataset, wrt a reference classification.

Processing pipelines including these activities can be defined in our DSL by a sequence of SQL-like statements.

For example, to apply POS-tagging to a set of sentences *dset* extracted from a text file, and select the sentences containing a verb, we could write:

```
create temporary table wdset as
  select x, pos_tag(x) from dset ;
create temporary table vbset as
  select sq from wdset
  where hasVerb(sq) ;
```

A subset of the SQLite.g4 Antlr grammar (857 LOC) is used for the NLP DSL. A CGTL script *nlp.cstl* is defined to translate DSL specifications to Python (with NLTK used for NLP process steps). This consists of 112 lines of CGTL code and took 2 person days to develop and test.

The CGTL rules to process the DSL statements and translate them to Python include:

```
select_stmt::
_1 |-->_1

simple_select_stmt::
_1 |-->_1

select_core::
select _1 , _2 from _3 |-->
  [_2 for _1 in _3]

select distinct _1 from _2 where _3 |-->
  set({_1 for _1 in _2 if _3})
select _1 from _2 where _3 |-->
  [_1 for _1 in _2 if _3]
```

For the above example, the result of executing the CGTL script is:

```
wdset = [pos_tag(x) for x in dset]
vbset = [sq for sq in wdset if hasVerb(sq)]
```

The DSL has been tested with a wide range of NLP pipelines including POS-tagging, chunking, entity extraction and the derivation of UML use cases from informal user stories. Table 1 shows the Python code generation time for the test cases of DSL models. The generated code size is 4KB in each case.

An alternative OCL-based NLP grammar (379 LOC) has also been developed[3], together with a CGTL file *nlp1.cstl* (89 LOC).

8.1.2 Mobile App Specification

This DSL provides facilities for defining classes with attributes, and stereotypes identifying the location

[3]Using the OCL grammar at https://github.com/antlr/grammars-v4/tree/master/ocl

Table 1: CGTL performance on NLP cases.

Case	*Model size*	*Generation time*
NLP Pipeline 1	12 LOC	0.3s
NLP Pipeline 2	17 LOC	0.4s
Use case derivation	22 LOC	0.3s
Entity extraction	22 LOC	0.3s
NLP Pipeline 5	24 LOC	1s
NLP Pipeline 6	24 LOC	2.8s
NLP Pipeline 7	12 LOC	0.85s
Regex matching	10 LOC	0.1s
Fuzzy similarity	12 LOC	0.7s
Average	17.3 LOC	0.75s

(remote or local) and persistence of the class data. Global functionalities of the system are defined as use cases. The DSL tools specified in CGTL scripts map the class definitions to SQLite databases (for local persistent classes) or to Firebase cloud datastores (for remote persistent classes) and data access objects to interface to these implementations. The use case definitions become operations of a business tier facade component, and are also used to produce SwiftUI views (screens) that trigger the business tier operations. Value objects are used to transfer business data between tiers. The DSL grammar is defined by the Antlr 4 grammar Mobile.g4 (274 LOC), and took 2 person days to write and test. The CGTL scripts consist of 15 separate scripts with a total of 1870 LOC and took 8 person days to write and test. The scripts are organised on the basis of the different target files which are generated: each script produces a specific component within the app architecture, such as a model facade, value object class, database interface or UI screen. To avoid duplication of common script tasks, such as mapping OCL expressions to Swift code, these are factored out into a separate script called from the component production scripts.

The DSL and tools were tested with a wide range of mobile app specifications including finance and health apps. Table 2 gives the size and code generation time for the test cases.

In comparison, a similar DSL definition using Xtext and Xtend took over 2 person months to develop, due to the high complexity involved in writing correct Xtext grammars, which combine parsing and semantic analysis. From a DSL grammar, Xtext produces a metamodel for the DSL, together with a DSL parser, however this dual interpretation of an Xtext grammar has deficiencies in terms of the quality of the metamodel and the reusability of the grammar (Izquierdo and Molina, 2014).

Overall we can conclude for **RQ1** that the combination of Antlr and CGTL is effective for defining text-based tools for DSL processing, since the effort required for writing grammars and CGTL scripts is

Table 2: CGTL performance on mobile app cases.

Case	*Model size (LOC)*	*Generated code size*	*CG time*
Person app 1	32	28KB	0.8s
Student app	60	39KB	1.47s
Person app 2	40	29KB	1s
Person app 3	46	30KB	1.2s
BMI app	15	14KB	1.1s
Health app	40	22KB	1.96s
Bond app	120	23KB	23.6s
Person DB app	34	32KB	6.6s
Student DB app	60	60KB	15.2s
Student app cloud + DB	60	50KB	7.5s
Average	50.7	32.7KB	6.05s

relatively low, and the efficiency of script execution is satisfactory for practical use. The expertise required to define CGTL scripts is (i) understanding of the source language grammar structure; (ii) understanding of the target platform syntax and semantics; (iii) knowledge of the CGTL syntax. There is no need to understand the metamodels of the source or target languages.

8.2 RQ2: Program Translation

To answer RQ2 we develop and evaluate example abstraction translations from Java 6/7, JavaScript, C, Cobol85 and Visual Basic 6 to UML/OCL. These are then applied to over 400 examples, including real-world programs from finance applications.

8.2.1 Translation Accuracy

We apply the Java 6/7 to UML/OCL abstraction mapping to 100 Java cases, consisting of 61 examples of Java library facilities (from java.io, java.lang, java.math and java.util), 34 examples of Java language features, and 5 cases of complete Java applications, including three cases taken from a package of financial software (bond valuation: Bondapp; yield curve computation: NSapp; CDO risk evaluation: CDOapp). We also applied the C to UML/OCL abstraction to 70 C examples, consisting of 14 statement examples, 36 declaration/type examples and 20 examples exercising the full range of standard C libraries. We applied the VB6 to UML/OCL abstraction to 100 VB6/VBA examples, including extracts from a large suite of bond pricing functions executed within Excel. We applied the Cobol85 abstraction to 85 cases, 64 concerning statements and 21 other language aspects. Finally, we applied the JavaScript to UML/OCL abstraction mapping to 85 JavaScript cases, consisting of 49 language construct examples and 36 data structure examples.

From the abstractions we then performed forward engineering to Python, Swift, C#, C++, Java 8, C and Go (in the case of the Java 6/7 examples), to Swift, C# and Go (for the C examples), to Python (for the JavaScript and VB6 examples) and Java (for Cobol85 cases).

To evaluate the correctness of the translations, we run equivalent tests on the original source program and translated target for each case, and compute the percentage of test results which agree. The same test values/parameters are used for both source and target programs. The test agreement percentage is shown in the cells of Table 3. There are a total of 439 tests for the Java 6/7 cases, 147 for the C cases, 197 for the VB6 cases, 183 for Cobol85 and 162 for JavaScript (JS). This measure of accuracy is the same as the *computational accuracy* measure used by (Lachaux et al., 2020).

Table 3: Evaluation cases: accuracy.

Target	*Source language*				
lang.	Java	C	JS	VB6	Cobol
Python	93%	–	95%	82%	–
Swift	96%	84%	–	–	–
C#	96%	90%	–	–	–
Go	90%	91%	–	–	–
Java 8	98%	–	–	–	88%
C++	93%	–	–	–	–
C	86%	–	–	–	–
Averages	93%	88%	95%	82%	88%

It can be seen that translations involving a large semantic distance between the source and target (such as C to Swift) are generally less accurate than those between similar languages (such as Java to C#). Nevertheless the accuracy is quite high, and for the industrial cases all numerical computations were translated without error. In contrast to these results, the java2python tool only achieves an accuracy of 38.3% on the Java to Python examples of (Lachaux et al., 2020), and transcoder achieves 68.7% accuracy, using 463 tests.

In terms of the efficiency of the translation process, the abstraction stage is comparable to code generation. Table 4 shows the time taken for this step, and for code generation, for the Java application cases. The Swift and J8 code generators use CSTL. The Python code generator is written in OCL.

8.2.2 Completeness

Completeness of the translation approach can be measured in terms of the percentage of the Java, JavaScript, VB6, Cobol85 and C grammar rules, including rule variants, which have corresponding abstraction rules in our reverse-engineering scripts.

Table 4: CGTL performance on Java 6/7 application cases.

Case	*Abstraction*	*CG Time (ms)*		
	Time (ms)	*Python*	*Swift*	*J8*
Bondapp	964	0	2032	256
app6	353	5	2783	219
NSapp	641	3	2022	220
CDOapp	1338	7	3683	1515
CorrCalc	4079	0	2443	198
Average	1475	3	2593	481.6

Table 5 shows the percentages of Antlr Java parser grammar rules which have corresponding abstraction rules, for each of the main syntactic divisions of Java. For C, 138 of the 153 grammar rules/cases of (Kernighan and Ritchie, 1988) are covered (90%), and 158 of 179 library operations (88%). For JavaScript, 258 of 324 grammar rules/rule options are covered (80%), and 39 of 53 library components. For VB6, 197 (86%) of the 229 statements, operations, functions and types of VB6 are represented. For Cobol85, 31 of 35 core statement kinds are covered (88%).

Table 5: Grammar rule coverage: Java.

Category	*Grammar cases*	*Abstraction rules*	*Coverage*
Types	33	31	94%
Expressions	101	82	81%
Statements	76	72	95%
Declarations	176	160	91%
Total	386	345	89%

Overall we can conclude for **RQ2** that the accuracy of the translation is high compared with other translation approaches, and that the computational efficiency is satisfactory. The completeness for each source language can be improved in principle by extending the coverage of language features and libraries. This may require the addition/extension of OCL libraries, such as the definition of a component to represent Excel functions and data in a platform-independent manner.

9 RELATED WORK

MDE usability issues have been highlighted by several studies of MDE in practice (Abrahao et al., 2017; Whittle et al., 2017). In response to these identified problems, different processes for using MDE have been proposed, such as the combination of agile methods with MDE (Alfraihi and Lano, 2017). Automated assistance for MDE processes, using AI techniques, has also been proposed (Burgueno et al., 2021). In particular, a promising approach for improving the usability of MDE is *modelling by-example* (Burgueno et al., 2019a; Lano et al., 2021; Lano and Xue, 2022). Applied to DSL engineering, this concept includes the automated learning of natural language to DSL mappings (Desai et al., 2016), and applied to model transformation and code generator synthesis, it enables the derivation of transformation/generator rules from relatively small datasets of examples (Lano et al., 2021; Lano and Xue, 2022). In the case of DSL tooling, the example-based process has the drawback that there may be ambiguous results for learnt mappings, which must be resolved by the user (Desai et al., 2016). For MT/code generator synthesis, there are limitations on the form of rules which can be learnt: in particular, CSTL rules involving actions cannot be learnt by the process of (Lano and Xue, 2022).

A transformation language related to CGTL is the Gra2MoL text-to-model language (Izquierdo and Molina, 2014). This provides a facility to search and extract information from ASTs, with benefits in terms of conciseness and clarity compared to conventional MT languages. CGTL can also define queries to navigate deeply into AST terms, using called functions as $\_i`f$ to test or extract information from a term bound to $\_i$. This approach is heavily used in the mobile app DSL (Section 8.1.2). Otherwise, GraMoL involves model manipulation in the style of ATL rules, and hence requires knowledge of the target metamodel.

Our program translation approach is related to reverse and re-engineering approaches which use a formal intermediate language, such as (Bowen et al., 1993; Liu et al., 1997). However, we use UML/OCL as the intermediate representation, instead of a formal specification language. This has the advantage of being more widely understood by software practitioners, and more widely supported by tools.

10 LIMITATIONS AND FUTURE WORK

The pattern-matching facilities of CGTL (on the LHS of rules) are limited to the immediate subterms of the input term. This could be extended, but would result in increased execution time.

With regard to DSL definition and processing, we restrict attention to textual DSLs. Currently our approach is geared to work with Antlr, but in principle other parsing technologies could be used.

Modern programming languages such as Java have extensive libraries and hence in these cases it is infeasible to model the semantics of the complete language including the libraries. Users of our translation tools may extend the CGTL abstraction scripts as required to add semantics for specific program libraries. Likewise, for COBOL, specific installations may use local extensions of COBOL facilities, for which a customised translation will need to be created.

An interesting area to pursue is the combination of symbolic and non-symbolic machine learning to learn program or language translations. Non-symbolic ML could be more effective than symbolic ML in learning large-scale translations involving thousands of special cases (eg., abstractions of Java library operations).

11 THREATS TO VALIDITY

Threats to validity include bias in the construction of the evaluation, inability to generalise the results, inappropriate constructs and inappropriate measures.

11.1 Threats to Internal Validity

11.1.1 Instrumental Bias

This concerns the consistency of measures over the course of the analysis. To ensure consistency, all analysis and measurement was carried out in the same manner by a single individual (the first author) on all cases. The comparison with the results of (Lachaux et al., 2020) used the same accuracy measure and a similar test-based evaluation approach to the evaluation in (Lachaux et al., 2020). Analysis and measurement for the results of Tables 1, 2 were repeated in order to ensure the consistency of the results.

11.1.2 Selection Bias

We chose mobile app specification and NLP specification as typical of the DSL modelling scenarios that arise in practice. The evaluation cases were designed to exercise all the significant choice points within the CGTL scripts. With regard to program translation, Java to C and JavaScript to Python are commonly requested translations (eg., on stackoverflow.com). COBOL to Java translation is of high significance to business. We selected example cases for evaluation of translation based on the grammars of the source languages, in order to cover the widest possible range of grammar constructs and options. As in (Lachaux et al., 2020), the translation examples are mainly cases involving single methods/functions, however some complete classes and complete real-world applications have also been analysed.

11.2 Threats to External Validity

11.2.1 Generalisation to Different Samples

As discussed above, our approach is restricted to textual DSLs and text-based processing, thus it is not directly applicable to graphical DSLs. We considered examples of the three main 3GL categories in our program translation work: Java represents the category of classical object-oriented languages, C, Cobol85 and VB6 represent the category of procedural languages, and JavaScript is representative of prototype-based languages with implicit typing. Thus a wide spectrum of programming languages has been considered, facilitating the generalisation of our work to other source languages in these categories, such as C# and Python. Other parsing tools could be used to produce ASTs, and other MDE tools such as Papyrus (Papyrus, 2022) used for forward engineering.

11.3 Threats to Construct Validity

11.3.1 Inexact Characterisation of Constructs

Our concepts of DSL processing and program translation are aligned to widely-used concepts in language engineering. We have given a precise characterisation of ASTs and CGTL via metamodels (Figures 2 and 3), and a detailed semantics of CGTL (Appendix).

11.4 Threats to Content Validity

11.4.1 Relevance

The Antlr and CGTL approach has been shown to be applicable to the processing of typical textual DSLs based on UML/OCL subsets or SQL-like notations, and to the processing of programming language source code for a range of languages, which include three of the most-popular programming languages according to the TIOBE index. Thus the approach should be relevant to other similar tasks in these domains.

In terms of usability, Antlr has an established history of use by numerous users over 15 years, and is actively maintained. AgileUML has been used for 20 years (prior to 2019 it was called UML-RSDS) in a wide range of educational, research and industrial applications, by users with varied skill levels. Thus there is evidence that these technologies are relevant to improving MDE usability.

Our experience with MDE adoption in industry (Section 2) indicated that grammar-based tools and simple DSLs were more acceptable and relevant to new adopters of MDE compared to metamodel-based tools and mathematical languages such as OCL.

11.4.2 Representativeness

The 3GL code translation tasks we have examined (translation of Java, C, VB6, Cobol85 and JavaScript) are representative of typical program translation tasks for 3GLs. Synthesis of executable code from DSL specifications is also a representative task for DSL processing.

11.5 Threats to Conclusion Validity

We used the concept of *computational accuracy* to measure the quality of program translations. This measure is also used by (Lachaux et al., 2020) and appears to be more appropriate to software translation than measures such as the BLEU score, used in machine translation of natural languages.

12 CONCLUSIONS

We have defined MDE techniques for DSL support and program translation which should be usable by general software practitioners. We have shown that the techniques can be combined to effectively perform typical language processing tasks such as generating code from DSL models and abstracting information from source code. We have also shown that certain kinds of model transformation rule can be expressed in CGTL.

The described approaches utilise lightweight tools (Antlr and AgileUML) which have low resource utilisation, and hence low environmental impact.

REFERENCES

Abrahao, S., Bourdeleau, F., Cheng, B., Kokaly, S., Paige, R., Stoerrle, H., and Whittle, J. (2017). User experience for MDE. In *MODELS 2017*.

Agarwal, M., Talamadupula, K., Martinez, F., Houde, S., Muller, M., Richards, J., Ross, S. I., and Weisz, J. D. (2022). Using document similarity methods to create parallel datasets for code translation.

Alfraihi, H. and Lano, K. (2017). The integration of agile development and MDE: a systematic literature review. In *Modelsward 2017*.

Antlr (2022a). https://www.antlr.org.

Antlr (2022b). Antlr StringTemplate, https://www.stringtemplate.org/about.html.

Bowen, J., Breuer, P., and Lano, K. (1993). A compendium of formal techniques for software maintenance. *IEE/BCS Software Engineering Journal*, 8(5):253 – 262.

Bucchiarone, A., Cabot, J., Paige, R., and Pierantonio, A. (2020). Grand challenges in MDE: an analysis of the state of the research. *SoSyM*, 19:5–13.

Burgueno, L., Cabot, J., and Gerard, S. (2019a). An LSTM-based neural network architecture for model transformations. In *MODELS '19*, pages 294–299.

Burgueno, L., Cabot, J., and Gerard, S. (2019b). The future of model transformation languages: an open community discussion. *JOT*, 18(3).

Burgueno, L., Clariso, R., Gerard, S., Li, S., and Cabot, J. (2021). An NLP-based architecture for the autocompletion of partial domain models. In *CAiSE 2021*, pages 91–106. Springer.

Buttner, F. and Gogolla, M. (2014). On ocl-based imperative languages. *Science of Computer Programming*, 92:162–178.

Clark, T. and Muller, P. (2012). Exploiting model-driven technology: a tale of two startups. *SoSyM*, 11:481–493.

Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., Sailesh, R., and Roy, S. (2016). Program synthesis using natural language. In *ICSE 2016*, pages 345–356.

Eclipse (2022). Eclipse OCL Version 6.4.0, https://projects.eclipse.org/projects/modeling.mdt.ocl.

Eclipse Agile UML project (2022). projects.eclipse.org/projects/modeling.agileuml, accessed 18.5.2022.

Izquierdo, J. C. and Molina, J. G. (2014). Extracting models from source code in software modernization. *Software Systems Modelling*, 13:713–734.

Kernighan, B. and Ritchie, D. (1988). *The C Programming Language*. Prentice Hall, 2nd edition.

Lachaux, M.-A., Roziere, B., Chanussot, L., and Lample, G. (2020). Unsupervised translation of programming languages. arXiv:2006.03511v3.

Lano, K., Kolahdouz-Rahimi, S., and Fang, S. (2021). Model Transformation Development using Automated Requirements Analysis, Metamodel Matching and Transformation By-Example. *ACM TOSEM*, 31(2):1–71.

Lano, K., Kolahdouz-Rahimi, S., and Jin, K. (2022). OCL libraries for software specification and representation. In *OCL 2022, MODELS 2022 Companion Proceedings*.

Lano, K. and Xue, Q. (2020). Agile specification of code generators for model-driven engineering. In *2020 15th International Conference on Software Engineering Advances (ICSEA)*, pages 9–15.

Lano, K. and Xue, Q. (2022). Code generation by example. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 84–92.

Lano, K., Yassipour-Tehrani, S., Alfraihi, H., and Kolahdouz-Rahimi, S. (2017). Translating from UML-RSDS OCL to ANSI C. In *OCL 2017, STAF 2017*, pages 317–330.

Liu, X., Yang, H., and Zedan, H. (1997). Formal methods for the re-engineering of computing systems. In *Compsac '97*.

Papyrus, E. (2022). Papyrus toolset, https://www.eclipse.org/papyrus.

Sanders, J. (2019). https://www.techrepublic.com/article/jpmorgans-athena-has-35-million-lines-of-python-code-and-wont-be-updated-to-python-3-in-time.

Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., and Heldal, R. (2017). A taxonomy of tool-related issues affecting the adoption of MDE. *Sosym*, 16:313–331.

Zaki-Ismail, A., Osama, M., Abdelrazek, M., Grundy, J., and Ibrahim, A. (2022). Rcm-extractor: an automated nlp-based approach for extracting a semi-formal representation model from natural language requirements. *AUSE*, 29(1):1–33.

Zhao, L. et al. (2020). Natural language processing for requirements engineering: a systematic mapping study. *ACM Computing Surveys*.

Zucker, J. and D'Leeuwen, M. (2020). Arbiter: a domain-specific language for ethical machine learning. In *AIES '20*.

APPENDIX

A CGTL script for processing a language $\mathcal{L}$ consists of a group of *rulesets*, each ruleset has a name and an associated sequence of rules, and processes syntax trees with the same tag as the ruleset name. Figure 3 shows the metamodel of CGTL, adapted from that of CSTL (Lano and Xue, 2020).

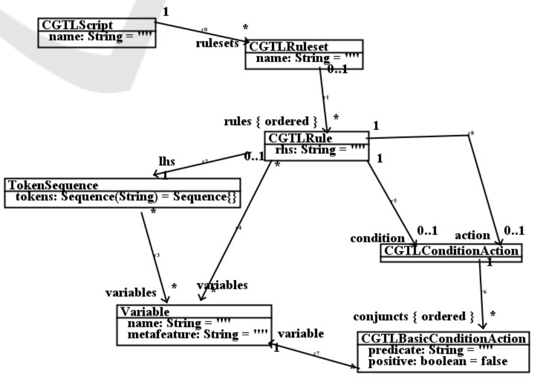

Figure 3: CGTL metamodel.

Ruleset names are usually the same as the source language syntax categories defined in the $\mathcal{L}$ grammar. A ruleset has the format:

```
rulesetName::
(rule\n)+
```

Individual rules in CGTL have one of the four forms:

```
lhs |-->rhs
lhs |-->rhs<when> Conditions
lhs |-->rhs<action> Actions
lhs |-->rhs<when> Conditions <action> Actions
```

Conditions can test the syntactic category of elements stored in CGTL variables, and other properties of these elements. They have the syntax

```
(variable value,)* variable value
```

where a *variable* is $\_i$ for integer i or $\_i`f$ for identifier f. The *value* is an identifier p or a negated predicate: *not p*.

The left hand side (LHS) of a rule is a schematic representation of textual concrete syntax in the source language $\mathcal{L}_1$, e.g., in a DSL or 3GL, and the right hand side (RHS) is the corresponding concrete syntax in the target language $\mathcal{L}_2$ which the LHS should translate to. The LHS represents a parse tree t of $\mathcal{L}_1$ elements, with its tokens corresponding to *t.terms* for composite t, or to *t.value* for basic t. Apart from literal text concrete syntax items, the LHS may contain variable terms $\_1$, $\_2$, etc, representing direct child subtrees of the tree (the properties of these elements can be constrained by the optional rule condition), and the RHS refers to the translation of these child elements also by $\_1$, $\_2$, etc. The special variable $\_*$ denotes a list of syntactic elements, and _$ represents a script parameter. Specialised rules are listed before more general rules. If no rule LHS matches a source parse tree then the tree content is copied to the output, this can be useful for debugging scripts.

One script g can be invoked from another, f, by the notation $\_i`g$ in a rule of f. Control is transferred to script g. In addition, the same notation can be invoked to apply a specific ruleset g within the same script to $\_i$. These facilities enable modularisation of CGTL code. Metafeatures such as the unparsed textual form of an element, or its inferred type, can also be referred to via $\_i`rawText$ or $\_i`type$. In comparison with Xtend, EGL or other template-based languages, there are no delimiters separating literal target language text from template language text. An <action> clause can be added to rules, to attach information to elements (eg., inferred types of elements) that can be read by subsequent rule applications. The *action* clause has the same syntax as the conditions clause.

CGTL semantics is based on text matching and substitution. The semantics defines the result $cgtl(s,t)$ of applying a CGTL script s to a composite or basic AST term t:

$$cgtl(s,t) = cgtl(s,tg,t)$$

if $t.tag = tg$ and s contains a ruleset tg::, otherwise $cgtl(s,t) = t$. For a ruleset name tg of s,

$$cgtl(s,tg,t) = rhs[t'_1,...,t'_n]$$

where $t.tag = tg$, each t'_i is $cgtl(s,t_i)$, and the rule

```
lhs[_1,...,_n] |-->rhs[_1,...,_n]<when>
                      Conditions[_1,...,_n]
```

of tg:: is the first whose *lhs* matches t and whose *Conditions* are true. If there is no such matching rule then $cgtl(s,tg,t) = t$.

t matches *lhs* if t's subterms *t.terms* match successive tokens of *lhs*: symbol terms of t must equal corresponding tokens of *lhs*, and non-symbol terms t_i are bound to corresponding variables $\_i$ in the token list of *lhs*. A variable $\_*$ binds to a list of successive *terms* which occur between specific symbol terms. r is then applicable to t if $Conditions[t_1,...,t_n]$ also hold. In this case, the script s is applied to each of the t_i to produce $t'_i = cgtl(s,t_i)$ if $\_i$ occurs as a simple variable expression on the RHS. However, if it occurs as $\_i`f$ for ruleset name f, then $t'_i = cgtl(s,f,t_i)$, and in the case of a script name g: $t'_i = cgtl(g,t_i)$.

A variable $\_*$ is replaced in *rhs* by the string concatenation of the t'_i of the terms bound to it. Various built-in functions such as *recurse* and *first* have specific denotations. Other metafeatures $\_i`f$ are evaluated as $ASTTerm.features[t_i + ""] \rightarrow select(x \mid x \rightarrow hasPrefix(f + " = ")) \rightarrow collect(y \mid y \rightarrow after(" = ")) \rightarrow any()$, in both the rhs and conditions.

If a rule with actions $Actions[\_1,...,\_n]$ matches term t, then the *Actions* are executed on the subterms t_i of t bound to the $\_i$. An action $\_i\ p$ adds p to $ASTTerm.features[t_i + ""]$ and an action $\_i$ *not* p removes p from $ASTTerm.features[t_i + ""]$. An action $\_i`f\ p$ adds $f + " = " + p$ to $ASTTerm.features[t_i + ""]$ and an action $\_i`f$ *not* p removes $f + " = " + p$ from $ASTTerm.features[t_i + ""]$.