Agile Specification of Code Generators for Model-Driven Engineering

Kevin Lano, Qiaomu Xue
Dept. of Informatics
King's College London, UK
Email: kevin.lano@kcl.ac.uk, qiaomu.xue@kcl.ac.uk

Shekoufeh Kolahdouz-Rahimi
Dept. of Software Engineering
University of Isfahan, Iran
Email: sh.rahimi@eng.ui.ac.ir

***Abstract*—The production of code or other text from software models is an essential task in Model-Driven Engineering (MDE) approaches for software development. Automated code generation is key to the productivity improvements observed in MDE approaches. Nonetheless, there has been a lack of systematic research into optimising the construction of code generators, and in the current state of the art such generators are usually developed manually, which involves detailed programming in 3GLs, or in specialised code generation languages. In either case, high expertise in the source language abstract syntax is necessary. In this paper, we survey different approaches for the construction of code generators, and we define an approach for declarative specification of code generators by text-to-text mappings, in terms of the concrete syntax of both source and target languages. We show that this approach enables the rapid development of code generators, which are also more concise and efficient compared to previous generators.**

Keywords — Code generation; Agile development; UML; Model-Driven Engineering.

I. INTRODUCTION

Code generation is the production of programming language code (e.g., Java) or other text (e.g., XML) from a model of a source language, such as a subset of UML, or a Domain-Specific Language (DSL). Code generation is an essential step in the application of Model-Driven Engineering (MDE) [18] to software application development, enabling the automated production of software artifacts from specification or design models, which are usually at a higher abstraction level than the artifacts (i.e., they abstract away from details of a specific implementation platform or programming language). This means that an application can be specified once and different code versions generated automatically from the specification, targeted at several different platforms. For example, a mobile app could be specified in a platform-independent form, and then separate implementations generated from the specification, targetted at the iOS and Android mobile platforms [14]. For a new target platform or language version, a new code generator needs to be produced, but existing application specifications can be reused.

Despite the importance of code generation, there has been relatively little research published on optimising the overall production process of code generators [5][25]. Instead, most published work has focussed on describing particular code generators [1][10][19][26], and issues specific to a particular generator. There are no general guidelines for assuring the quality of code generators, and perhaps as a consequence of this lack, the quality of automatically generated code is sometimes poor in comparison with manually-written code [13][17].

The task of developing a code generator consists of three main activities:

1) Defining a semantically valid representation of the source language in the target language and verifying this representation;
2) Defining a code generation strategy to create the target representation of each source model;
3) Writing and testing code generation rules in a 3GL or code generation language.

If we restrict attention to source languages which are subsets of UML, crucial concepts which need to be represented in a target language are *classes and interfaces*, *features* (attributes, references and operations), *inheritance and polymorphic operation semantics*, *object creation/deletion*, *object communication*, *object state changes*, and *Object Constraint Language (OCL) data types and operators*.

The task of finding a valid representation may be relatively simple if there is small semantic distance between UML and the target (e.g., in the case of OO programming languages, such as Java, C# and C++, which support orthodox class/inheritance concepts). But for other target languages (e.g., for C, Python, JavaScript) there is considerable semantic distance from UML, and no simple encoding of UML concepts.

In this paper we focus on items 2) and 3) above. Section II reviews the state of the art in code generation approaches, and Section III introduces our approach to code generation specification and implementation, using the $\mathcal{CSTL}$ concrete syntax transformation language. Section IV provides a comparative evaluation of the approach, Section V discusses threats to validity, and Section VI gives related and future work.

II. CODE GENERATION APPROACHES

A code generator can be defined either in terms of the *abstract syntax* of the languages it relates, or in terms of

 ISBN: 978-1-61208-827-3

their *concrete syntax*: abstract syntax refers to the conceptual elements of a language, represented as a Backus-Naur Form (BNF) grammar or metamodel, and the features and interrelations and constraints of these elements. Concrete syntax refers to the written textual form or graphical form of models/programs or other artifacts in a language.

For example, part of the metamodel of the C language is shown in Figure 1.

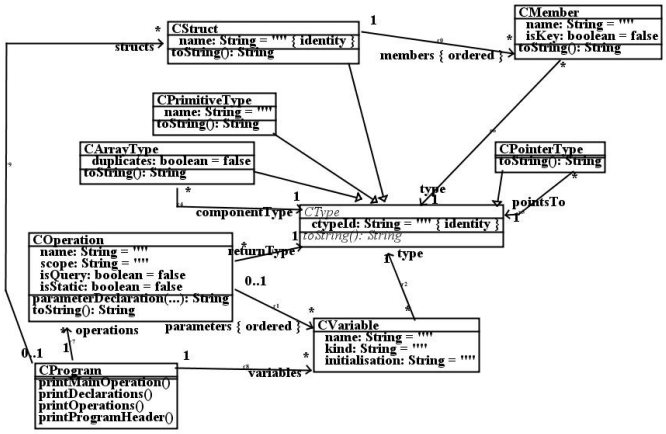

Figure 1. C language metamodel

The *CStruct* class is the abstract syntax representation of C `struct` type definitions, which have the concrete syntax:

```
struct S
{ MT1 cm1;
  ...
  MTn cmn;
};
```

where each of the `MTi cmi;` are the concrete syntax of *CMember* elements that are the members of the *CStruct*.

In these terms there are three general approaches to writing code generators:

1) *Abstract syntax to abstract syntax*: code generation rules are written in a programming language or in a general Model Transformation (MT) language, to map from elements of the abstract syntax of the source language, to elements of the abstract syntax of the target language.
2) *Abstract syntax to concrete syntax*: templates using target concrete syntax are defined, with variable parts or slots within these templates having content depending upon source model elements and expressions written in terms of source language abstract syntax.
3) *Concrete syntax to concrete syntax*: the mapping is defined only in terms of source and target language concrete syntax, with rules specifying how source concrete syntax fragments should be represented in target concrete syntax.

In cases where abstract syntax is used, this involves navigation over the elements of a source or target model using the data structures of a 3GL, or via an expression language, which is usually based on OCL [24].

An example of the first approach is the UML to C code generator of [19], which is written in OCL. The second approach is supported by specialised model-to-text MT languages, such as EGL [8], Acceleo [2], and Xtext/Xtend [9]. The third approach is supported by concrete syntax MT languages, such as the concrete graph transformation language of [12].

A. Issues in code generation

The task of code generation may require the resolution of several fundamental problems:

1) *Language abstraction gap*: the distance between the source and target languages is too great to be bridged by a single generation step, and multiple steps or human input is needed. For example, declarative specification in Agile UML [6] is expressed using OCL predicates as invariants or postconditions. This specification is quite distant from executable code, and several translation stages are required to refine the specification into an executable form.
2) *Structural gap*: the source and target languages may have different structures, so that one source element can contribute to the target syntax of multiple target elements (a 1-many relationship), or information from multiple source elements could be combined to form one target element (a many-1 relationship). For example, a UML class attribute is represented by a C struct member, and by possibly multiple setter and getter operations, in separate header and code files [19].
3) *Semantic gap*: where source language concepts cannot be directly represented in the target language because of semantic differences, so that a more complex and indirect encoding is needed. For example, UML inheritance has no direct representation in C [19], and differs in its semantics from Python or JavaScript inheritance. Likewise, the constructs of UML statemachines have no direct correspondence with conventional programming language constructs [1][26].

Each of the three general approaches listed above address these problems in different ways, and there are also various tradeoffs in each approach. Abstract syntax to abstract syntax specifications can address the language abstraction gap by using intermediate languages between the source and target, and chaining transformations in sequence (Figure 2). Structural and semantic gaps can be addressed by complex coding in terms of the abstract syntaxes. However this

 ISBN: 978-1-61208-827-3

approach requires deep understanding of the metamodels of both source and target languages, and understanding of OCL-style navigation expressions, in addition to knowledge of target language concrete syntax. Because abstract syntax is typically more verbose and detailed than concrete syntax, such specifications can become very large and complex programs or transformations, which are not easy to understand or maintain.

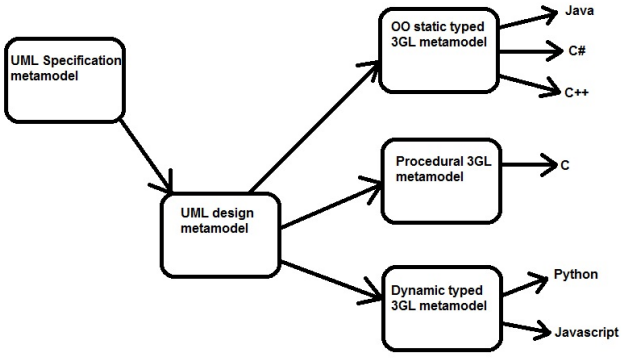

Figure 2. Staged code generation

Abstract to concrete syntax approaches using template languages involve specification with a combination of abstract syntax source language expressions and concrete target text. Thus to use these approaches, knowledge of the source language metamodel and of the target language concrete syntax is needed, but not of the target language metamodel. Again, complex navigation expressions are typically needed to refer to and select source model elements. The structure of a template-based specification is usually closely tied to the structure of the target language program components.

To address language gaps, preliminary model-to-model transformations could be used as in Figure 2, with only the final step being model-to-text [15]. The preliminary transformations could construct a design model suitable for direct translation to any target language in a given language family. For structural and semantic gaps, auxiliary operations and data would need to be used, with possibly multiple passes through the source model, and multiple output templates (e.g., for C production from UML, a C header file template and a C code file template would be needed). This again requires complex specification in the source abstract syntax. The mix of source and target languages in one artifact can be confusing, and such specification can be prone to errors due to misuse of delimiters between the language texts [21].

Concrete syntax to concrete syntax approaches have the advantage that no knowledge of abstract syntax is needed. In addition, any navigation over source and target models is implicit, based on the concrete syntax structures. Thus, the code generation rules can be defined in an intuitively natural manner in terms of the concrete source and target syntax. However, for cases of abstraction, structural or semantic gaps, more complex mapping mechanisms are needed, with auxiliary operations and possibly multiple rules/multiple passes over the source text. In principle, concrete syntax to concrete syntax transformations could be chained as in Figure 2 to use intermediate textual languages to bridge gaps.

III. Concrete syntax to concrete syntax specification using $\mathcal{CSTL}$

Our experience of building large code generators in Java [20] and in OCL [19] convinced us that a more usable, agile and lightweight approach was needed.

For the simpler specification of code generators, we have developed a textual concrete syntax transformation notation $\mathcal{CSTL}$. This is a DSL for code generation, which enables the direct definition of code-generation transformations by means of concrete syntax to concrete syntax mappings.

A. $\mathcal{CSTL}$ concepts

$\mathcal{CSTL}$ is designed to be a small language, which can be used by general software practitioners, and does not require a high degree of MDE expertise. Its execution semantics can be understand in terms of familiar string matching and replacement concepts. Figure 3 shows the metamodel of $\mathcal{CSTL}$.

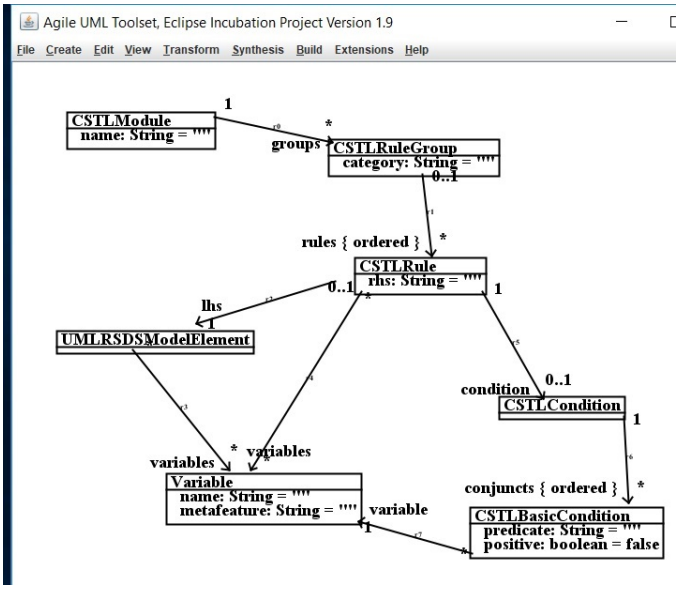

Figure 3. $\mathcal{CSTL}$ metamodel

A $\mathcal{CSTL}$ module consists of a sequence of rules grouped into categories. Individual rules in $\mathcal{CSTL}$ notation have the form:

```
selement |-->telement<when> Condition
```

The <*when*> clause and condition are optional. The left hand side (LHS) of a $\mathcal{CSTL}$ rule is some piece of concrete syntax in the source language, e.g., in Kernel Metamodel (KM3) [16] textual notation for UML class models, and the right hand side (RHS) is the corresponding concrete syntax in the target language (e.g., C, Java, Swift, etc), which the LHS should translate to. Apart from literal text concrete syntax, the LHS may contain variable terms _1, _2, etc, representing arbitrary source concrete syntax fragments (possibly constrained by the optional condition), and the RHS may refer to the translation of these fragments also by _1, _2, etc. This enables $\mathcal{CSTL}$ mappings to be applied recursively. Rules are grouped into source language syntactic categories, such as binary expressions or statements, and apply to elements in these categories. Specialised rules are listed before more general rules.

For example, to map a KM3 text syntax of a UML class to type and data declaration text in a C header file, assuming that the class contains only simple data feature definitions $x : T$, we could write the following rules to translate UML types, attribute declarations and class declarations:

```
Type::
Integer |-->int
Real |-->double
Boolean |-->unsigned char
String |-->char*
Set(_1) |-->_1*
Sequence(_1) |-->_1*
_1 |-->struct _1*<when> _1 Class

Attribute::
_1 : _2; |-->  _2 _1;\n
_1 : _2; _3 |-->  _2 _1;\n  _3

Class::
class _1 { _2 } |-->struct _1\n{_2};
class _1 extends _2 { _3 } |-->
struct _1\n{ struct _2* super;\n_3};
```

These rules translate a class declaration

```
class Customer extends Person
{ name : String;
  age : Real;
}
```

into:

```
struct Customer
{ struct Person* super;
  char* name;
  double age;
};
```

Rule conditions can be combined by conjunction (comma) and negation, for example:

```
_1 = _2 |-->_1 == _2<when>
  _1 not String, _1 not object,
  _1 not collection
```

Negation can often be avoided by using the ordering of rules. The above rule could alternatively be expressed as a default case after specific = rules for strings, objects and collections.

Any stereotype of an LHS model element may also be used as a condition on it, for example:

```
Attribute::
_1 : _2 |-->  let _1 : _2<when>_1 readOnly
```

for a mapping from UML to Swift.

The *metafeature* notation _*i*‘*f* enables access to features *f* of the abstract syntax. For example, _1‘*elementType* returns the element type of whatever language element is held in variable _1.

Furthermore, a set of rules can be grouped together in a single file, representing one way of mapping source elements to target elements. Separate files can define alternative or additional mappings. For example, separate files *cheader.cstl* and *cbody.cstl* could be used to create the header and body files of a C program derived from a UML model. This addresses the issue of 1-many structural gaps, and enables context-dependent alternative translations of source syntactic elements. If *f.cstl* is a file containing a $\mathcal{CSTL}$ module, then the notation _1‘*f.cstl* applies this module to the contents of _1.

B. $\mathcal{CSTL}$ semantics

The execution semantics of $\mathcal{CSTL}$ is based on string pattern matching and rewriting. Given a source text element *elem* of syntactic category *CT*, the first *CT* rule whose LHS matches *elem* and whose conditions are true is applied to *elem*, with metavariables _*i* of the LHS being bound to corresponding source fragments within *elem*. These fragments are then themselves mapped by the rule set and the result of transformation is substituted for _*i* on the RHS of the rule. If no rule applies, an element is mapped to itself.

Formally, a $\mathcal{CSTL}$ specification module *cg* contained in a file *cg.cstl* defines a function τ_{cg} from the texts of source language $\mathcal{L}_1$ to those of target language $\mathcal{L}_2$ as follows.

For each source text $e \in \mathcal{L}_1$, of $\mathcal{L}_1$ category *C*, successive rules *r* of the *C* :: group in *cg* are checked to determine if *r.lhs* can match to *e*.

Each *r* is of the form

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

 ISBN: 978-1-61208-827-3

An absent condition is interpreted as *true*. r matches e if e equals $r.lhs$ with substitutions of subtexts e_i of e for the variables $\_i$ of $r.lhs$, i.e., e equals $r.lhs[e_i/\_i]$ (ignoring leading or trailing spaces). r is then applicable to e if $Conditions[e_i/\_i]$ also hold. In this case, the result of cg applied to e is

$$\tau_{cg}(e) \;=\; r.rhs[\tau_{cg}(e_i)/\_i]$$

where r is the first C :: rule matching e. If no rule of the C :: group matches e, then e is copied to the output:

$$\tau_{cg}(e) \;=\; e$$

in either case, the specifier must ensure that the result $\tau_{cg}(e)$ is a valid text of $\mathcal{L}_2$.

For example, the UML attribute declaration `s : Sequence(Real);` matches the LHS `_1 : _2;` of the first *Attribute* rule above, with `_1` bound to `s` and `_2` bound to `Sequence(Real)`. The latter type text is then rewritten to `double*` by the *Type* rules for *Sequence* and *Real*, so that the overall result of the attribute rule application is `double* s;\n`.

Metafeatures $\_i$‘f are evaluated as $\tau_{cg}(e_i\text{‘}f) = \tau_{cg}(\overline{e_i}.f)$, where $\overline{e_i}$ is the abstract syntax element corresponding to e_i.

A module application $\_i$‘$f.cstl$ is evaluated as $\tau_{cg}(e_i\text{‘}f.cstl) = \tau_f(e_i)$.

C. $\mathcal{CSTL}$ applications

We have applied $\mathcal{CSTL}$ to the translation of UML to Java (UML2Java8), and to Swift (UML2Swift). These are used as part of UML to Android and UML to iOS mobile app generation tools. $\mathcal{CSTL}$ is also provided as part of the Agile UML toolset [6] as a facility to enable users to quickly specify new code generators from UML to different programming languages, hence extending the toolset for their own needs.

The emphasis in the UML2Java8 and UML2Swift code generators is on the generation of fully functional code from OCL expressions. OCL has over 100 operators [24], thus at least these many rules are needed to translate OCL expressions to program code. 152 of the 183 rules (83%) of the UML2Java8 generator are either mapping rules for different kinds of OCL expression (139 rules) or for statements (13 rules). Some examples of expression rules from this generator are:

```
BinaryExpression::
_1 & _2 |-->_1 && _2
_1->count(_2) |-->Collections.frequency(_1,_2)
_1->select(_2 | _3) |-->
  Ocl.selectSet(_1,(_2)->{return _3;})
    <when> _1 Set

_1->includes(_2) |-->_1.contains(_2)
_1->includesAll(_2) |-->_1.containsAll(_2)
```

A Java 8 library of OCL functions, Ocl.java, is also defined to support the implementation of some operators, such as $\rightarrow$*select*. The UML2Swift generator is similarly constructed.

$\mathcal{CSTL}$ can also be used for general DSL to code synthesis, provided that the DSL elements can be expressed as a subset of our UML source language. Stereotypes can be used to label UML elements as representing DSL elements, and rules for DSL code generation expressed in terms of these stereotypes.

IV. Evaluation

We evaluate the approach by comparing UML to 3GL code generators specified using different approaches (Table I). We compare the development effort of the generators, and their sizes, syntactic complexity and efficiency. The size is measured in lines of code (LOC). *MaxES* is the maximum OCL expression size used in navigation expressions (operators + identifiers in the expression). This is the *MEL* measure of [28]. All approaches cover the structural parts of the generated code, but differ in how they synthesise behaviour (constructor and method bodies).

TABLE I. COMPARISON OF CODE GENERATION APPROACHES

Generator	*Implemented*	*Size*	*MaxES*	*Scope*
UML2C++ [20]	Java	18,100	–	Behaviour from OCL
UML2Java [7]	Acceleo/ Java	3,957	27	Outline behaviour
UML2Java [27]	EGL	1,425	35	Statemachine behaviour
UML2Java8 [6]	$\mathcal{CSTL}$	426	11	Behaviour from OCL
UML2Swift [6]	$\mathcal{CSTL}$	398	5	Behaviour from OCL

Table I shows that the $\mathcal{CSTL}$ Java generator is substantially smaller compared with other UML to Java approaches. Unlike the Acceleo and EGL generators, it is focussed on behaviour implementation instead of structural implementation. The declarative nature of the specification should also make it easier to comprehend and to change than imperative or hybrid code generators involving explicit model navigation. The syntactic complexity is indicated by the maximum condition or navigation expression size – for the $\mathcal{CSTL}$ solution this is also significantly smaller than for the other solutions.

In Table II we compare the performances of the Acceleo, $\mathcal{CSTL}$ and a Java-coded UML to Java code generator on the Acceleo test model *example.uml*, which consists of 6 classifiers, 8 data features, 6 operations and 5 inheritance relations (Figure 4).

Larger examples were created by duplicating this basic structure. We also added some functionality to the operations of the example. The results in Table II show that the $\mathcal{CSTL}$

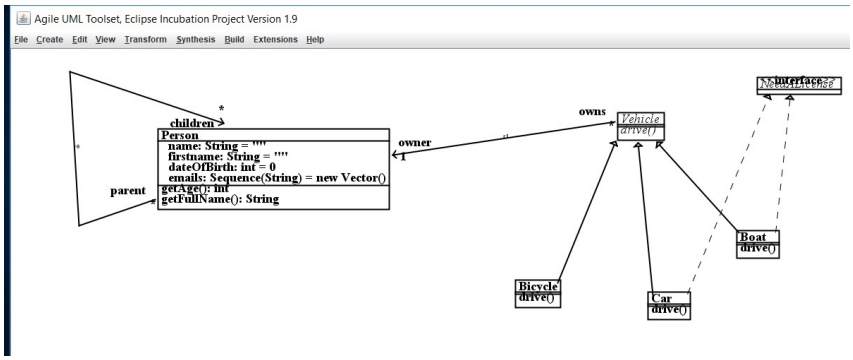

Figure 4. Acceleo example model

generator is approximately 10 times more efficient than the Acceleo generator, despite including OCL expression processing. This improvement is likely to be caused by the purely tree-based processing of $\mathcal{CSTL}$, in contrast to the graph navigations of Acceleo. $\mathcal{CSTL}$ specifications contain no global variables or other 'memory' of which rules have been applied, so that rule applications (e.g., upon separate classes) are independent and could be parallelised. The $\mathcal{CSTL}$ solution is similar in efficiency to the Java-coded UML2Java4 generator of [6].

TABLE II. PERFORMANCE COMPARISON OF UML TO JAVA CODE GENERATION APPROACHES

Model	*Size*	*Acceleo*	*$\mathcal{CSTL}$ (UML2Java8)*	*Java (UML2Java4)*
1	25	480ms	45ms	28ms
2	50	750ms	70ms	47ms
4	100	800ms	36ms	37ms
10	250	1.12s	80ms	79ms
15	375	1.52s	104ms	194ms

TABLE III. COMPARISON OF CODE GENERATOR DEVELOPMENT EFFORT

Approach/Generator	*Effort*
Acceleo/Java: UML2Java [7]	3000+ hours
Java: UML2C++ [20]	2170 hours
OCL: UML2C [19]	1375 hours
$\mathcal{CSTL}$: UML2Java8	36 hours
$\mathcal{CSTL}$: UML2Swift	50 hours

Table III compares the development effort of different generators, in terms of person hours. These show substantial reductions in effort for the $\mathcal{CSTL}$ developments, compared to developments using 3GL programming, templates or OCL. This reduction arises because (i) the code generator file has a modular structure based on the source language syntax categories; (ii) no programming language or OCL code needs to be written; (iii) the overall size of the generator is substantially reduced and is contained in 2 or 3 small files. Not only is the initial effort lower in the $\mathcal{CSTL}$ generators, but also the cost of making changes to the specification. We can also compare the size of the generated code for the example model of Figure 4: this is 107 LOC for the $\mathcal{CSTL}$ UML2Java8 generator, 380 LOC for the Acceleo UML2Java, and 1628 LOC for the Java coded UML2Java4 translator of [6]. The latter provides many additional functionalities, such as input and export of models from XML and CSV, which the $\mathcal{CSTL}$ and Acceleo generators do not.

V. THREATS TO VALIDITY

We address *instrumental bias* by performing all measurements in a consistent manner. Regarding *selection bias*, our evaluation example is taken from the Acceleo repository, and hence it is independent of the authors. Regarding generalisation from the single example presented here, we have also used $\mathcal{CSTL}$ to generate app code for several Android and iOS apps of different kinds, and found similar results in terms of high efficiency and low generated code size. Regarding *relevance*, we have only implemented UML to code mappings, DSL to code mappings are the subject of future work.

VI. RELATED AND FUTURE WORK

Investigations into the use of machine learning, specifically Long Short Term Memory (LSTM) neural nets, to synthesise model transformations from examples of input and output models have shown that this can be practically useful [4], however it requires large numbers of examples and significant training time. The same approach could potentially be used to derive code generators from examples presented either in abstract or concrete syntax.

Tools have also been created that convert UI sketches into UI code [3][22]. These are based on object recognition approaches, so could be used as a means of processing manually-drawn concrete syntax graphical models (such as class diagrams or activity diagrams) prior to code generation from these models. Other low-code approaches for code production are template-based or data-based app builders, such as Microsoft PowerApps [23] or Google AppSheet [11].

 ISBN: 978-1-61208-827-3

A disadvantage of ML approaches, such as LSTM neural nets, are that they only produce implicit 'black box' specifications of generators. In contrast, $\mathcal{CSTL}$ specifications provide a clear and explicit expression of how source language syntax maps to target language syntax. Compared to app builders, we use a wide range of UML features to define application data and functionality. The $\mathcal{CSTL}$ rules give precise control over which code elements are produced for these specifications.

An important area for future work is ensuring the quality of generated code [13]: code generators should not increase the technical debt burden of a software system, and where possible should ensure that code quality standards are met. There should not be unnecessary code generated, and duplicated and excessively complex code should be avoided, together with other flaws, such as bidirectional module dependencies. Our approach can avoid complex duplicated code by factoring out complex code definitions into the OCL support library. An example is the complex Swift code needed for regular expression matching: the Ocl.swift library defines a function $matches(str : String, pattern : String)$, which encapsulates this code, so that the $\mathcal{CSTL}$ translation rule can be simplified to:

```
_1->matches(_2) |-->
  Ocl.matches(str: _1, pattern: _2)
```

VII. Conclusion

We have considered alternative approaches for the definition of code generators, and proposed a novel declarative approach, which permits simpler and more concise specifications, compared to existing approaches. We showed that the approach can produce smaller and more efficient code generators for UML to Java transformation.

References

[1] A. Aabidi, A. Jakimi, R. Alaoui and E. Hassan El Kinani, "An object-oriented approach to generate Java code from hierarchical-concurrent and history states", Int. Journal of Information and Network Security, vol. 2, 2013, pp. 429–440.

[2] Acceleo project, https://www.eclipse.org/acceleo, accessed 18.8.2020.

[3] T. Beltramelli, "pix2code: Generating code from a GUI screenshot", https://arxiv.org/abs/1705.07962, 2017. Accessed 18.8.2020.

[4] L. Burgueno, J. Cabot, and S. Gerard, "An LSTM-based neural network architecture for model transformations", MODELS '19, 2019, pp. 294–299.

[5] A. Dieumegard, A. Toon, and M. Pantel, "Model-based formal specification of a DSL library for a qualified code generator", OCL 2012, pp. 61–62.

[6] Eclipse Agile UML project, https://projects.eclipse.org/projects/modeling.agileuml, accessed 18.8.2020.

[7] Eclipse UML2Java code generator, https://git.eclipse.org/c/umlgen/, accessed 18.8.2020.

[8] Epsilon project, https://projects.eclipse.org/projects/modeling.epsilon, accessed 18.8.2020.

[9] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way", OOPSLA 2010, pp. 307–309.

[10] M. Funk, A. Nysen, and H. Lichter, "From UML to ANSI-C: an Eclipse-based code generation framework", RWTH, 2007.

[11] Google, https://www.appsheet.com, accessed 18.8.2020.

[12] R. Gronmo, B. Moller-Pedersen, and G. Olsen, "Comparison of three model transformation languages", ECMDA-FA, 2009, pp. 2–17.

[13] X. He, P. Avgeriou, P. Liang, and Z. Li, "Technical debt in MDE: A case study on GMF/EMF-based projects", MODELS 2016, pp. 162–172

[14] H. Heitkotter, T. Majchrzak, and H. Kuchen, "Cross-platform MDD of mobile applications with MD^2", SAC 2013, ACM Press, 2013, pp. 526–533.

[15] Z. Hemel, L. Kats, D. Groenewegenn, and E. Visser, "Code generation by model transformation: a case study in transformation modularity", Sosym, 9: 375–402, 2010.

[16] F. Jouault and J. Bezivin, "KM3: a DSL for metamodel specification", ATLAS team, INRIA, 2006.

[17] L. Kapova, T. Goldschmidt, S. Becker, and J. Henss, "Evaluating maintainability with code metrics for model-to-model transformations", QoSA 2010: Research into Practice – Reality and Gaps, Springer, 2010, pp. 151–160.

[18] K. Lano, *Agile model-based development using UML-RSDS*, CRC Press, 2016.

[19] K. Lano, S. Yassipour-Tehrani, H. Alfraihi, and S. Kolahdouz-Rahimi, "Translating from UML-RSDS OCL to ANSI C", OCL 2017, STAF 2017, pp. 317–330.

[20] K. Lano, H. Alfraihi, S. Kolahdouz-Rahimi, M. Sharbaf, and H. Haughton, "Comparative case studies in agile MDD", FlexMDE 2018, MODELS 2018, pp. 203–212.

[21] K. Lano, S. Fang, H. Alfraihi, and S. Kolahdouz-Rahimi, "Simplified specification languages for flexible and agile modelling", FlexMDE, MODELS 2019, pp. 460–467.

[22] Microsoft, *sketch2code*, https://www.microsoft.com/en-us/ai/ai-lab-sketch2code, accessed 18.8.2020.

[23] Microsoft, *PowerApps*, https://powerapps.microsoft.com, accessed 18.8.2020.

[24] OMG, *Object Constraint Language Specification v2.4*, 2014.

[25] I. Stuermer, M. Conrad, H. Doerr and P. Pepper, "Systematic testing of model-based code generators", IEEE TSE, 33(9), 2007, pp. 622–634.

[26] E. Sunitha and P. Samuel, "Object-oriented method to implement the hierarchical and concurrent states in UML state chart diagrams", Software engineering research, management and applications, Springer-Verlag, 2016, pp. 133–149.

[27] TU/e, SLCOtoJava1.0 code generator, https://gitlab.tue.nl/SLCO, 2020.

[28] M. Wimmer, S. Martinez, F. Jouault, and J. Cabot, "A Catalogue of Refactorings for model-to-model transformations", Journal of Object Technology, vol. 11, no. 2, 2012, pp. 1–40.

 ISBN: 978-1-61208-827-3