Electronic Communications of the EASST
Volume 82 (2022)

11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation

-

Doctoral Symposium, 2022

Towards Code-centric Code Generators

Daniel Busch

13 pages

Guest Editors: Sven Jörges, Salim Saay, Steven Smyth
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Towards Code-centric Code Generators

Daniel Busch

Chair for Programming Systems, Faculty for Computer Science, TU Dortmund University

Abstract: This paper presents a novel approach to code generation. While common code generator approaches lack in support for code evolution and maintenance such as refactoring, the presented Code-centric generator (CCG) approach attempts to overcome these issues. Instead of mixing generator abstractions and actual code snippets, CCG provides a layer between the generator and prototypical target code. The new layer provides the ability to map code generator operations directly onto code AST subtrees, and generates the resulting generators based on these mappings and the prototypical target implementation.

Keywords: Code Generation, Abstract Syntax Tree, Model-driven Engineering, Domain-specific Languages, Tooling

1 Introduction

Code generation is an important part of Model-driven Engineering (MDE) [Völ09, Sel03]. It is widely used to transform (graphical) models into executable code or structured data formats [Fow10]. There are different approaches for different purposes, but each approach has some major drawbacks, so no current solution is truly satisfactory.

Especially in the context of full code generation [KT08], generated code can become quite large and complex. Like any other software, generated code must be maintainable and evolvable as requirements may change over time. Template-based generators are probably the most commonly used generators for full code generation. They are strongly focused on the desired target code they are supposed to produce. Many parts of the resulting code are hard-coded, and dynamic parts that should be filled with data extracted from MDE models are explicitly marked as such. Template-based generators scale well because they are easy to develop. However, most approaches are difficult to maintain and evolve as new or different requirements arise. This problem is caused by the fact that hard-wired pieces of code are strings representing parts of the desired code rather than parsable pieces of code. As a consequence it is often not possible to use common development tools that allow refactoring, linting, code styling, static analysis, or similar.

In order to exploit the advantages of template-based generators and to overcome their disadvantages, this paper proposes the approach of Code-centric Generators (CCG).

This paper will first provide a more in-depth view of traditional template-based generators, their capabilities, advantages, and also their drawbacks in Section 2. Next, Section 3 presents the newly developed CCG approach, which works on parsable code and its abstract syntax tree (AST) instead of simple string representations. After that, an example is used to illustrate the differences in generator development for template-based generators and Code-centric Generators, as well as the similarities and differences in their structure and overall properties in Section 4.

2 Common Code Generator Approaches

Code generation is prevalent in the MDE landscape. Most modeling tools offer the ability to generate code from modeled instances. Full code generation solutions even offer the ability to generate entire programs, which can be quite large and complex, so code generation must scale to meet the demands of such scenarios. The following paragraph introduces the template-based generation approach, so that the next section can highlight the differences to the CCG approach presented in this paper.

Template-based Generators An output-based way to generate code is the template-based approach [SLS18]. This approach is most commonly used when larger generation needs arise, such as in full code generation environments. The generators consist of static parts and dynamic parts. Static parts are hard-coded fragments of the desired target, that should be generated. To take full advantage of the generator approach in terms of using the data present in the generator's source, dynamic parts are used. These parts are intertwined with the static parts in the form of placeholders, statements, or expressions. All of them are used to either bring data of the source into the desired generation target or to modify the structure of the generation target depending on the source.

Famous examples of template-based generators are Xtend string templates [Bet16], FreeMarker[1], or Velocity[2]. All of these examples use their own expression languages that can be used alongside static strings to produce arbitrary code.

The main advantage of template-based generators is the simplicity of their implementation. The implementations are language agnostic, so developers do not need to use the same language as the source and target. In addition, because of their static parts, which are only supplemented by dynamic parts, the overall implementation steps resemble "common" software development processes of the target.

At the same time, this focus on the target may also be the greatest disadvantage of some approaches. By adding dynamic parts, static parts are no longer parsable or executable as instances of their target language. This leads to problems when maintaining or evolving the code for further development. Commonly used tools such as linters, refactoring tools, or other checkers do not work for template generators. As a result, desired changes can lead to round-trips, as their changes must be checked for different source inputs. Undiscovered resulting structures that are the result of edge-case source inputs may not be discovered, leading to unexpected problems and errors. While this is a problem for the template-based generator approaches mentioned above, it is not the case for every approach. Thymeleaf, for example, uses natural templates to overcome this problem [3]. These templates add dynamic parts to static parts of the template while still maintaining correct syntax. Only when the generation process is triggered will the dynamic parts be processed and result in substitutions in the resulting output. However, this approach is only viable for a small set of targets, as it must be designed around the underlying syntax. In the case of Thymeleaf, natural templates are offered for HTML, JavaScript, and CSS.

[1] https://freemarker.apache.org
[2] https://velocity.apache.org
[3] https://www.thymeleaf.org

3 Code-centric Generators

While template-based generators have proven feasible for large projects and full code generation, they may lack proper maintainability and evolution support because existing tools are not applicable to generator templates. To overcome this problem, this paper proposes Code-centric Generators (CCG), which try to eliminate the drawbacks of template-based generators by generating them from prototypical code implementations instead of manually implementing the templates directly. Making the CCG approach a generator for code generator templates.

CCG focuses on generating from prototypical implementations of the desired target. But instead of discarding this prototype and manually translating it into template generators, CCG uses prototypes as a central artifact not only for the creation of generators, but also for further evolution and maintenance.

To accomplish this, CCG relies on the following:

1. A prototype of the desired target. This is a concrete instance of one of the target outputs that the resulting generator should be able to create. As a result, by adding dynamic parts, this prototype should be able to produce every feature and every aspect that possible target outputs are intended to contain.

2. A CCG meta-description. Besides a reference to the prototype (e.g. in the form of an absolute file path), this description file must contain the mappings of generator operations to AST subtrees. Details on this mapping are described later.

3. A tool for creating the meta-description and generating the resulting generator or template. Since some parts of CCG rely on parsing an AST, computing IDs, and generating code (in the form of the code generator), the entire CCG approach relies on tool support. Similar to the reliance of the template-based approach and its syntax, which has to be realized as an internal or external DSL. As the description of such a tool is not central to the overall idea behind this approach, more about this tooling can be read in Section 6.

In its simplest form, the CCG meta-description just points to a prototype code file without any additions. This would allow to generate exactly the code contained in the prototype. However, since generators are usually intended to enrich code with information extracted from an underlying model, additional data can be provided to modify the prototype code. Although the data is not provided in the CCG meta-description, the necessary operations, that are comparable to the dynamic part of template-based generators, can be specified. For this purpose, CCG allows for references to arbitrary nodes of the prototype's AST nodes. Each node is assigned an ID that is computed based on the node's path from the root of the AST to the node. To modify the prototype for the resulting generator, CCG references to AST nodes (i.e. subtrees) can be one of four different operations:

1. **Substitutions** should be replaced by arbitrary input data in the resulting generated code. To achieve this, an input is expected to be inserted instead of the selected AST node and its subtree.

2. **Deletions** allow subtrees of the AST to be omitted.

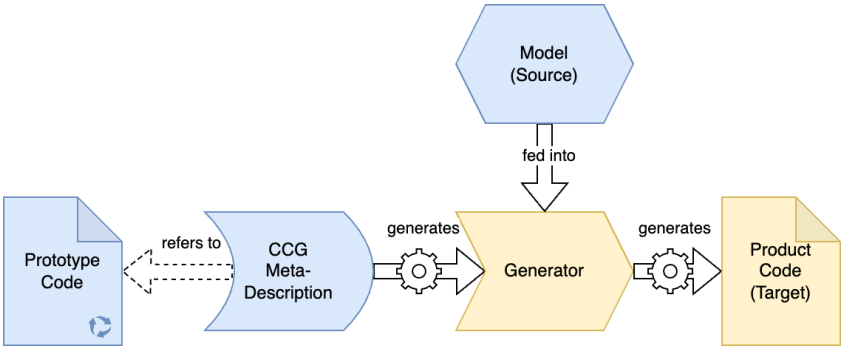

Figure 1: Overview of CCG development artifacts

3. **Conditions** lead to the expectation of boolean statements. Depending on these statements, the subtree of the selected node is either included in the generated target or not.

4. **Repetitions** can behave in one of two ways. First, they can behave similarly to conditions when no node in its subtree contains another substitution of condition markings. In this case, they expect a numeric value indicating how many times their subtrees should be repeated in the resulting target.

   Second, they may also contain further substitutions or condition markings in their subtree. Accordingly, they expect a collection of data that fits all subtree operations. The number of repetitions of the subtree then depends on the number of entries in the provided collection. This allows parts of the prototype code to be repeated for recurring patterns in the underlying model from which the data is extracted.

Based on the aforementioned reference to the prototype code, the computation of IDs for AST nodes, and the mapping of the presented operation to node IDs, it is possible to generate template generators that accept data according to the mapped operations and enrich the modeled template to produce the desired target code.

Since the computation of the node IDs and thus the mapping of the operations is a rather abstract task, tool support is crucial for CCG. These tools should accept paths to prototype code files, parse them, assign IDs to each AST node, and also allow users to map operations to each node. Furthermore, the generation process of the resulting generator should also be handled by the same tool, taking into account the data provided.

Figure 1 shows all the artifacts that are part of the CCG workflow. It is apparent that the prototype code is not just a by-product of the workflow, but rather a central artifact that the CCG file references and from which the generator is created. Blue colored artifacts are parts that can be edited during evolution and maintenance. Changes to these artifacts are reflected in the resulting generator and thus in the generated product code. The latter two, colored yellow, remain untouched at all stages of development and are only products that are used. The following section provides an example to give a further insight into the CCG approach and its artifacts.

4 Example & Comparison

The following section covers a simple example of generating a minimalist HTML web page that contains some images and captions. Generators for this example will be presented in a traditional template-based approach and also using the CCG approach. This small example will be enough to show some of the problems that can occur with a template-based generator and how CCG can eliminate these problems.

Scenario In this example, the generators should generate a simple HTML web page that contains only some images and corresponding captions. Generators should therefore expect an arbitrary collection (e.g. list, array, set, etc.) containing objects that have an attribute for the image path and another attribute for the corresponding captions.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>EASST Example</title>
5    </head>
6    <body>
7      <figure class="img-caption-block">
8        <img src="/path/to/img1" />
9        <figcaption>
10         Lorem ipsum dolor sit amet
11       </figcaption>
12     </figure>
13     <figure class="img-caption-block">
14       <img src="/path/to/img2" />
15       <figcaption>
16         Consectetuer adipiscing elit
17       </figcaption>
18     </figure>
19   </body>
20 </html>
```

Listing 1: Example expected target HTML

After using the generators to create our desired target web page, we want to make sure that our generated HTML meets some conditions. We want to use an HTML linter to make sure that no HTML syntax is violated and also that each "img" tag has an "alt" attribute for better accessibility. Of course, for a small and simple example like this, checking these conditions can easily be done by hand, but one can easily understand that this is tedious and error-prone for larger scenarios. Especially for full code generation scenarios, where entire applications are generated, automated tools are essential to maintain quality standards.

The following paragraphs cover this example. First, a template-based generator is used to generate the described target. Then the same problem is solved using the CCG approach. In addition to creating the generators, the example covers the use of a linter as described earlier. Finally, the advantages and disadvantages of both approaches are compared.

Template-based Generator When developing a template-based generator, it may be beneficial to first create a prototypical target program. In this example, the prototype is the code shown in Listing 1. To create a template-based generator from this prototype, developers typically identify which parts of the code should remain static and which need to be filled dynamically. The dynamically populated parts are fed data from the underlying model for which the target is to be generated.

For the target described earlier, it is easy to see that one of the blocks with the class IMG-CAPTION-BLOCK must be repeated for each input entity, while the other block with the same class can be omitted. Thus, the block will probably be wrapped in a control sequence that allows for iterations and repetitions depending on the input data. Within this block, the value of the image source must also be dynamic. Therefore, the value of the source should be replaced by an expression that inserts data from the entity of the current iteration step. The same should be done for the content inside the caption tag below the image.

An example implementation of such a template is shown in Listing 2 in the form of an Xtend string template with imaginary entities and properties.

```
1  '''<!DOCTYPE html>
2  <html>
3          <head>
4              <title>EASST Example</title>
5          </head>
6          <body>
7              «FOR entity : imageCaptionPairs»
8                  <figure class="img-caption-block">
9                      <img src="«entity.imgSrc»" />
10                     <figcaption>
11                         «entity.caption»
12                     </figcaption>
13                 </figure>
14             «ENDFOR»
15         </body>
16 </html>'''
```

Listing 2: Example Xtend string template

After creating the template, a linter should be used to check some properties of the code. In this case, as mentioned earlier, it should be ensured that each image tag also has an ALT attribute to improve accessibility. Unfortunately, it is not possible to use the linter on the template, since its representation with the embedded dynamic parts is not parsable.

Consequently, the linter must be used on the prototype to check for the desired properties. After checking for and possibly adding the missing attributes, the code generator developer has to repeat the earlier steps of identifying the static and dynamic parts, and introducing the structural and insert expressions. These additional steps, which must be done multiple times, result in round-trip overhead.

This example is quite small and simple, so this could have been done manually. However, this may not be the case for more complex generators or projects where many generators are used, such as full code generation projects.

Not being able to use the linter because the template is not parsable is a problem that applies to

almost any supporting tool that requires parsing. None of these tools, such as refactoring helpers, static analysis checkers, and others, are usable with these templates.

```
{
    "prototypePath": "/path/to/prototype",
    "operationMappings": [
    {
            "operationType": "deletion",
            "nodeId": "d2e16e6"
        },
        {
            "operationType": "repetition",
            "nodeId": "e78f543"
        },
        {
            "operationType": "substitution",
            "nodeId": "a98931d"
        },
        {
            "operationType": "substitution",
            "nodeId": "87d4eeb"
        }
    ]
}
```

Listing 3: Example CCG file

Code-centric Generator The code-centric approach uses the prototype in Listing 1 as a central development artifact. For this purpose, the prototype should be stored along with all other source code files. A newly created CCG file will store the path to the prototype in order to resolve a reference to it. The CCG file will also contain the operation mapping information and forms a new abstraction layer between the prototype code and the actual generator that will be generated in a later step. Similar to the template-based approach, developers should first identify which parts of the prototype are static and which parts are dynamic. Instead of replacing the dynamic parts with expressions in some kind of template, the desired operations are mapped to the nodes that have been identified as dynamic. In this case, the subtree of the second IMG-CAPTION-BLOCK block element is mapped to the deletion operation, the first IMG-CAPTION-BLOCK block element is mapped to a repetition operation, and the value of the image source and the content of the paragraph tag are mapped to substitution operations. The mapping is done by associating the specified operations with previously computed node IDs. The IDs should be computed with tool support, see Section 6 for more information. Listing 3 shows an example CCG meta-description in JSON format. Although the JSON is human-readable, it is not possible for humans to see which operation is mapped to which AST node. Therefore, Section 6 proposes some graphical representations that allow operations to be edited and visualized directly in more practical

representations of the AST.

The resulting CCG file now contains the same information about the static and dynamic parts of the generator as the template shown in Listing 2. The generator for the CCG approach can now be generated. This should also be done with tool support (see Section 6). The result should be a generator very similar to the template-based generator.

If we now want to use a linter to guarantee certain properties of the HTML, this can easily be done. The linter can be applied to the prototype and the missing ALT attribute can be added automatically. When the developer rebuilds the CCG generator, the new attribute is also added to the generator and no information about dynamic code is missing. The introduction of a new abstraction layer has made the prototype an essential part of the generator development, allowing developers to use any tool that requires parsable code. This advantage eliminates round-trips and opens up new possibilities for the evolution and maintenance of code generators.

Comparison The two examples show similarities and differences in the development of code generators and especially in the later lifecycle regarding code evolution and tool support.

Presumably, both approaches start by building a prototype to be sure of how the target will be built. However, only CCG uses this prototype as a central artifact for development. Template-based approaches may discard this prototype once the final code generator has been created. Both approaches also need to identify which parts of the code generator should be static and which should be dynamic. Template-based generators then wrap these parts into the desired template syntax (e.g. Xtend string template syntax in this example). CCG instead specifies the operations needed for the dynamic parts in a separate file that contains mappings of these operations to the prototype's AST nodes.

By using the prototype as a central artifact, CCG allows developers to use tools they are accustomed to. Any change to the prototype can also be reflected in CCG code generators, by simply regenerating the resulting code generator. If developers want to use supporting tools with template-based approaches, they have to use them on the prototype (if it has not been omitted) and make sure to propagate the changes manually. This can result in significant round-trips.

In addition to better tool support, CCG also allows for better testing. Template-based generators require output to be executable, and therefore must be used to generate exemplary instances of the generated code to enable testing. CCG, on the other hand, allows testers to utilize the prototype used for the generator to run their tests. Since the prototype must contain every feature that arbitrary outputs of the generator should be able to contain, tests can also be more certain to cover all possible instances of the outputs. With template-based generators, testers must also trust the completeness of the exemplary instances.

Figure 2 shows the artifacts of the template-based generator approach. The red colored prototype code may have been discarded, while the blue colored artifacts are edited during evolution and maintenance. The colors used are the same as in Figure 1, so that both approaches can be easily compared.

While this simple example illustrates the benefits of CCG, there are also some challenges that need to be investigated. Section 6 identifies some challenges and provides a brief outlook on how they might be addressed.

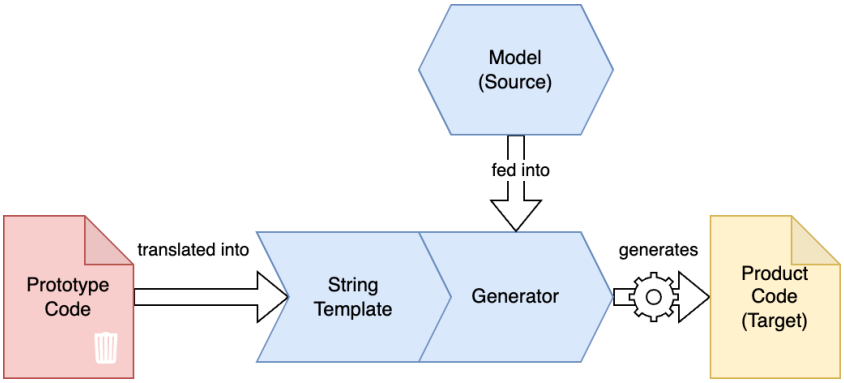

Figure 2: Overview of template-based generator development artifacts

5 Related Work

In addition to the code generation approaches already presented, there are several disruptive and innovative approaches. All of these alternative approaches have different advantages and, like CCG, try to eliminate drawbacks of existing techniques. This section describes two of these alternative approaches: Genesys [Jör13] and Machine Learning (ML)-based approaches. Genesys is covered because some of its aspects are similar to CCG, while ML-based approaches seem to be promising for the future.

Genesys An approach to code generation with a focus to service orientation and graphical modeling is Genesys [Jör13]. It strictly separates the output description of generators and the logic of the generator. The former is realized as services that can be different code generators, e.g. template-based or rule-based generators. The latter is part of a graphical DSL that allows to execute the code generator services or other code generation related tasks.

By using code generation services, Genesys is agnostic to any code generator paradigm. In addition, it provides a holistic solution because it also covers code generator evolution and maintenance, including test and verification utilities.

On the downside, Genesys is an entirely new tool that developers must learn from scratch. Unlike CCG, which aims to enable reuse of existing tools that developers are familiar with, Genesys provides a whole new tooling ecosystem that takes time to get used to.

Machine Learning-based approaches In the recent past, ML-based approaches to code generation have emerged. However, they have been used less for MDE and full-code generation. Popular tools such as GitHub Copilot[4] or similar are more often used to generate small snippets of code to assist developers. These tools take natural language descriptions of problems as input, consider the current context within the source code, and propose code snippets to the developer [NN22].

Approaches like GitHub Copilot are still relatively new and have yet to be used in large scale

[4] https://github.com/features/copilot

contexts. Nevertheless, they seem to be very promising for the future, also for MDE and full-code generation.

On the contrary, they suffer from the same problems as many ML applications: lack of explainability and trust, especially since deep learning techniques are mostly considered black boxes [XUD+19]. Generating whole applications with ML approaches can be problematic because developers may have to trust the resulting generate without any guarantees about its robustness or correctness.

6 Future Work

This paper presents the main ideas behind the CCG approach. There are several things that need to be done to realize this approach and make it a viable solution. In addition, there are some challenges in the CCG design that may be resolved after further investigation. This section covers the future work on the roadmap for future development and research around CCG.

Tool Reference Implementation Section 3 implies that the CCG approach requires tool support in order to be usable. There are three reasons for this:

1. ID calculations for AST nodes
2. Operation Mapping to these calculated IDs
3. Providing a generator for the resulting target code generator

Since the IDs require parsing the AST and then computing the IDs according to each node's path, this is only feasible if this is done in an automated manner by a supporting tool. The same is true for mapping operations to these IDs. While it is not important for users of the tool to know which ID they are assigning to which operation, it is still very important to be able to visualize which subtree has been mapped to which operation. Ideally, this can be done by showing users a graphical representation of the AST (e.g., in the form of a structured textual representation of the source code) and labeling selected subtrees with color-coded indicators. Figure 3 shows two example visualizations as they could be realized in a CCG tool. The left visualization is based on an AST-tree representation, while the right visualization is based on the concrete syntax and AST-mappings are made on top of this textual representation. Minimalist indicators seem to be a way to visualize the mappings well with as little distraction as possible from the usual source code representation.

Finally, it should be easy to generate the desired code generators. Although it is possible to provide the generators in a different way (e.g. as an executable script that takes the mappings file and the prototypical source code), it seems to be the best way to include the generation process in the tool that is needed anyway.

To easily add support for as many languages as possible, it is best to use a widely adopted parser. ANTLR [PQ95] seems to be a viable option for this. Future work should investigate whether ANTLR is suitable, and how it can be used to provide a simple interface that allows easy integration and deployment of new prototype languages.

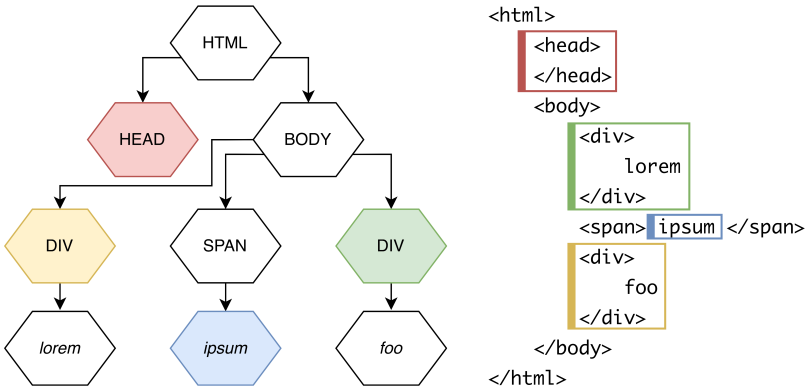

Figure 3: Two example visualizations of operations mapped to HTML

Maintaining Mappings after Prototype Evolutions One of the key challenges that CCG tries to solve is how to better evolve code generators in constantly evolving projects. Section 4 provides an example of how easy it is to make additional changes to the prototypical code that are propagated to the resulting code generators. However, there are also cases where the evolution of the prototypical code can lead to broken mappings and thus to potential loss of information. Three types of edits can lead to these mapping losses:

1. Changes between the root and an operation-mapped node
2. Edits to the operation-mapped nodes themselves
3. Deleting operation-mapped nodes

Any editing may result in changes to the structure of the AST. Changes between the root and a node of the AST will result in a new ID for that node. This can result in lost mappings because the mapped IDs may no longer exist. A possible workaround for this problem is to introduce a second ID, which is computed from the path of the (leftmost) leaf of the node's subtree. This additional ID can act as a repair mechanism so that the original root ID can be recomputed and reassigned. Future work will investigate the feasibility of this approach. The effectiveness of this fix-up depends on whether edits to both the root path and the leaf path occur regularly or not. If not, the fix-up will be of great value. Otherwise, the effect of the extra ID may be marginal.

Edits to operation-mapped nodes themselves lead to the same problem, but additional IDs cannot fix this. Alternatively, this problem may be solved by looking at the IDs of the parent and the leftmost child of the node. The feasibility and impact of looking at surrounding nodes for this problem also needs to be investigated in future work.

Finally, deletions can also result in lost references. Although this problem seems to be less problematic since the code evolution omitted the mapped-to node, it could still be confusing and clutter the mapping file if there are dead references. In addition, a deletion could easily be mistaken for one of the above problems with changes that are reflected the node's IDs. In both cases, all potential IDs are missing, but surrounding IDs may still exist.

Evaluation in real-world projects The true benefit of this new approach to code generation can only be explored when it is applied to real-world projects. Ideally, exemplary projects would be subject to frequent changes so that the main benefits of code evolution and maintenance can be examined. There are also plans to integrate CCG code generation into Cinco [NLKS18] and Cinco Cloud [BBK+22], two workbenches for building graphical modeling IDEs. The latter workbench and its resulting IDEs are entirely available on the web.

While real-world projects are certainly a part of future work, their use depends on first creating the aforementioned reference tool implementation. In addition, it would be beneficial to first address the aforementioned challenges of maintaining the operation mapping after prototype code evolutions. Only with these prerequisites can the full potential of CCG be properly explored.

7 Conclusion

This paper introduces CCG, a novel approach to code generation. CCG places the prototype created during code generator development as a central artifact of development, code evolution, and maintenance. By adding an additional abstraction layer, developers can reference a prototype and map typical code generator operations to nodes of the prototype's AST. CCG tools can use this abstraction layer to generate a code generator that is capable of generating target files as expected.

This novel approach leads to improvements in code evolution and maintenance over typical template-based generators. A major reason for this is that developers can now use arbitrary tools (e.g., linters, code checkers, refactoring tools) that require parsable code directly on the prototype. Changes made to the prototype are also reflected in the resulting code generator and thus in the generated code. Manual changes to the code or testing are also easier to accomplish, since they can be performed on the prototype as well.

However, the tool support required for CCG is also its main drawback. Instead of a universal approach that is available for virtually any target language, CCG relies on tool support for each target language individually.

While CCG is a very promising approach, its real benefits need to be further investigated. In addition to addressing some of the challenges mentioned above, a reference tool as well as an evaluation of the tool and the CCG approach need to be done in the near future.

Bibliography

[BBK+22] A. Bainczyk, D. Busch, M. Krumrey, D. S. Mitwalli, J. Schürmann, J. Tagoukeng Dongmo, B. Steffen. CINCO cloud: a holistic approach for web-based language-driven engineering. In *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering: 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22–30, 2022, Proceedings, Part II*. Pp. 407–425. 2022.

[Bet16] L. Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[Fow10] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[Jör13] S. Jörges. *Construction and evolution of code generators: A model-driven and service-oriented approach*. Volume 7747. Springer, 2013.

[KT08] S. Kelly, J.-P. Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[NLKS18] S. Naujokat, M. Lybecait, D. Kopetzki, B. Steffen. CINCO: a simplicity-driven approach to full generation of domain-specific graphical modeling tools. *International Journal on Software Tools for Technology Transfer* 20:327–354, 2018.

[NN22] N. Nguyen, S. Nadi. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*. Pp. 1–5. 2022.

[PQ95] T. J. Parr, R. W. Quong. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25(7):789–810, 1995.

[Sel03] B. Selic. The pragmatics of model-driven development. *IEEE software* 20(5):19–25, 2003.

[SLS18] E. Syriani, L. Luhunu, H. Sahraoui. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures* 52:43–62, 2018.

[Völ09] M. Völter. Best practices for DSLs and model-driven development. *Journal of Object Technology* 8(6):79–102, 2009.

[XUD+19] F. Xu, H. Uszkoreit, Y. Du, W. Fan, D. Zhao, J. Zhu. Explainable AI: A brief survey on history, research areas, approaches and challenges. In *Natural Language Processing and Chinese Computing: 8th CCF International Conference, NLPCC 2019, Dunhuang, China, October 9–14, 2019, Proceedings, Part II 8*. Pp. 563–574. 2019.