Survey of Special Purpose Code Generators

Marco Craveiro <marco.craveiro@gmail.com>

Abstract

MDE is an approach to software engineering centred around modeling. Code generators are an important component of the MDE toolset because they bridge the gap between models and their implementation. Studies have demonstrated that deficiencies in MDE tooling are a common cause of failure in MDE adoption, highlighting the need for research to identify and address inadequacies.

This paper has two contributions. First, it performs a survey of Open Source tools dedicated to special purpose code generation. Second, it identifies a core set of properties that characterise their approach, which we argue have application to code generation in MDE. These properties can be leveraged to guide new tool development as well as for evaluating existing tooling.

1 Introduction

Model Driven Engineering (MDE) is an approach to software development that places an emphasis on *modeling* and *models*.[1] One of the main tenets of MDE is a focus on automation, which is thought to increase development speed and enhance software quality, amongst other benefits [Völ+13]. These are claims supported by empirical evidence.[2]

Code generators are a key tool in this regard, since, as Jörges *et al.* state, they "[...] automatically derive an implementation from the model, and which thus relate to models in the same way in which compilers relate to high-level languages." [JMS08] Correspondingly, a great deal of research has been carried out with regards to their development and, in particular, their place within the MDE domain.[3]

Nonetheless, adoption studies point out that MDE has yet to move away from its niche status in Software Engineering [Mus+14], implying that more remains to be done in order to gain industry acceptance. Whilst appropriating responsibility for the current state of affairs, Whittle *et al.* uncovered a variety of complex factors impacting MDE adoption [Whi+17] — many of which non-technical — but confirmed that tooling has had a key part to play. The situation is perhaps best summarised by their dictum: "[m]atch tools to people, not the other way around" — spelling out a need to understand how MDE tools should work from the viewpoint of prospective users rather than the experts in the field who are developing them, and revealing a deficit in usability and Human-Computer Interaction (HCI) research. Where usability efforts have been made, they typically have focused on graphical modeling tools and the Unified Modeling Language (UML).[4] Clearly, from a usability perspective, the requirements of a modeling tool are very different from those of a code generator, so this paper argues instead for a corresponding separation of concerns between code generation and modeling. If this separation is accepted, then the established practices in the wider field of code generation — *i.e.* outside of MDE — become available as a source of lessons and best practices for MDE tooling. Start-

[1] MDE is member of a family of closely related approaches that share a *model-driven* focus, leading Völter to group them under the moniker of *MD** [Völ09]. The present paper refers only to MDE for the sake of simplicity, but the argument made is believed to be relevant to MD* in general. The interested reader is directed to Brambilla *et al.* [BCW12] for a broader treatment of MD*.

[2] As an example, whilst performing an assessment of MDE in industry [Hut+11], Hutchinson *et al.* reported a positive impact in maintainability and productivity, which, according to their respondents, was attributable to code generation. However, the study also elaborated on the difficulty of performing a clear and unambiguous impact assessment.

[3] Jörges *et al.* provide a good overview of the state of the art of code generation in [JMS08]. For an understanding of how it fits in the wider map of model transformations, see the feature model developed by Czarnecki and Helsen [CH06].

[4] The reader is directed to Yosser *et al.* [EA+15] and Harald [Stö14] for a sample of these efforts.

ing from this premise, the objective of the present study is to identify and analyse one such source — "special purpose code generators".

From an MDE point of view, the primary contribution of this paper is towards identifying a set of themes that can guide the development of future MDE tools — specifically code generators, but some of its elements may find application elsewhere — or act as a "checklist" for the evaluation of existing tools. A secondary contribution, which may be of use outside MDE, is the surveying of a set of special purpose code generators, as well as defining the terminology more precisely.

The remainder of the paper is structured as follows. Section 2 contains a literature review of related topics and their relationship with the present work. Section 3 describes the code generation landscape and defines special purpose code generators. It is followed by Section 4, which details the methods used in the survey and its limitations. Section 5 presents the tools that are the subject of our study. Section 6 identifies the core themes that emanated from the surveyed tooling and, finally, Section 7 summarises the paper and proposes areas for future work.

2 Related Work

Several studies have pointed out the importance of tooling to the outcome of MDE adoption. Of these, two are of particular relevance to this paper as they identify usability as an important direction of future research, as well as contextualising its significance within the broader challenges of adoption.

In [Mus+14], Mussbacher *et al.* outline a list of major current problems in MDE, where they include "Obstacles for Tool Usability and Adoption", stating: "Even after a suitable language and tool have been identified, the users face significant usability challenges, *e.g.*, steep learning curves, arduous user interfaces, and difficulty with migrating models from one version of a tool to the next." The paper then goes on to analyse — and propose solutions to — the broader problems with MDE, curtailing its relevance to the present work.

These difficulties are echoed by Whittle *et al.* in the aforementioned study [Whi+17]. As part of their intriguing diagnostic of usability and tool design, they state: "Most MDE tools are developed by those with a technical background but without in-depth experience of human-computer interaction or business issues. This can lead to a situation where good tools force people to think in a certain way." In addition to this call to arms towards usability research, the paper also defines a taxonomy of tool-related issues, a subset of which is used by this paper.

Finally, a number of experience reports related to MDE tooling development were analysed, and these did contain brief incursions on the topic but lacked sufficient focus on usability for the needs of this paper. In [PV12], Paige and Varró perform a detailed study of the lessons learned building model-driven tools but provided only allusions to usability concerns rather than direct analysis.[5] A similar pattern emerged with other reports [And+14; SKSG07].

In summary, this study is motivated by our findings in the literature. On one hand there is a clear need for more research on usability within MDE tooling, as pointed out by the adoption studies. On the other hand, code generation has been bundled together with more general modeling activities even though they have different usability requirements. The present paper addresses this gap by focusing exclusively on code generation, and attempts to identify and analyse the approaches of tools which may have applicability for MDE tooling.

3 Context

Code generation has historically been associated with automatic programming, and both have a long recorded history in Computer Science [Par85]. Whilst the term "code generator" lacks formal definition in Computer Science, informally, it is used to describe any tool that processes a well-defined input and generates "code".

"Code" may itself have different meanings, depending on context: within compiler engineering, it typically represents the binary *machine code*, whereas in the broader context of programming, it

[5] As an example: "In particular, the collaborators required a textual interface to any tools (the intended users preferred a textual interface instead of a graphical one). It was also perceived that a textual interface, and textual MDD languages, were preferred for fine-grained tasks such as specifying how models were navigated, evaluating expressions, etc." [PV12]

usually represents the textual *source code*, conforming to the grammar of a programming language. The present paper is only concerned with a subset of the latter: *special purpose code generators*. The next sections describe what is meant by this term by contrasting it with the more general notion of code generation in MDE.

3.1 Narrow Focus

MDE research expanded and generalised informal notions by framing code generation as an instance of a class of Model-to-Text (M2T) transforms[6], leading Brambilla *et al.* to state [BCW12] that, "[...] in MDE, code generation is the process of transforming models into source code". From this perspective, code generation is one of potentially several steps of a chain of model transformations required to produce a running system, and the design and implementation of code generators exist as part of the broader development activities that include the creation of Domain Specific Languages (DSLs) and the refinement of models at different levels of abstraction — from platform independent to platform specific representations. Thus, the MDE practitioner makes use of a plethora of code generation technologies and techniques[7] and integrates those with other modeling tools to meet specific code generation requirements. These tools and techniques provide the flexibility required for model-driven software development — at the expense of increased complexity — and so we categorise them as *general purpose* code generation tooling because they are designed to be adapted to open-ended requirements.

A very different use of code generation is made by a class of *special purpose* tools, typically designed for a single, well-defined objective. These tools tend to focus on domains such as XML serialisation support, generation of ORM for relational databases, binary serialisation of data structures and the like, all of which are functions of a structural definition. In contrast to the open-ended approach promoted by MDE tooling, these special purpose tools usually generate code not meant for modification — in cases, not even inspection — and with a limited and well-defined use.

As a representative example, Protocol Buffers[8] — a serialisation framework for structured data — states in its documentation (*emphasis theirs*):

> Protocol buffer classes are basically dumb data holders (like structs in C); they don't make good first class citizens in an object model. If you want to add richer behaviour to a generated class, the best way to do this is to wrap the generated protocol buffer class in an application-specific class. [...] *You should never add behaviour to the generated classes by inheriting from them.* This will break internal mechanisms and is not good object-oriented practice anyway." [Goo18a]

Thus, these special purpose tools are designed to satisfy the requirements of one use case only.

It is important to note that the ideas ascribed above to special purpose code generators are not entirely new within MDE — though the packaging may be. As an example, the term *cartridge* has been used to denote a similar concept though, arguably, a lack of a formal definition hindered its spread.[9]

3.2 Constrained Variability

Another viewpoint from which to contrast these two approaches is that of variability, where we can make use of Groher and Völter's work [GV07]. Though not a necessary condition, special purpose code generators typically support *structural variability* — that is, the creative construction of arbitrary data structures — but are often designed

[6] See Czarnecki and Helsen [CH03; CH06] for a detailed treatment of M2T transforms. Note that these were originally known as Model-to-Code (M2C) transforms, but the word "text" was preferred over "code" because the output of a M2T transform need not be source code — *e.g.* JSON, XML, *etc.*

[7] Many of which are detailed in Rose *et al.*'s feature model [Ros+12].

[8] https://developers.google.com/protocol-buffers

[9] In [Völ+13], Völter *et al.* states that "a cartridge is a 'piece of generator' for a certain architectural aspect". However, in [Völ09], Völter elaborates on his concerns for the term, and these are quite damning: "[I]t's not clear to me what it [a cartridge] really is. A cartridge is generally described as a 'generator module', but how do you combine them? How do you define the interfaces of such modules? How do you handle the situation where to cartridges have implicit dependencies through the code they generate?"

to restrict variants of structural models quite aggressively, when at all allowed.[10] MDE takes the opposing view by treating it as an important concern, giving rise to concepts such as negative and positive variability and to techniques for handling them. [GV07; GV09]

In practice, these are not binary opposite views. A more suitable way to describe the code generation landscape is as a *spectrum* of possibilities with regards to their purpose and take on variability, as Figure 1 illustrates, with each approach representing different kinds of trade-offs over factors such as complexity and flexibility.[11] Whilst a clear simplification, the visualisation nonetheless helps the intuition that there are choices to be made and alludes to the existence of useful traits of special purpose code generators which may be worth taking into account when developing MDE tooling.

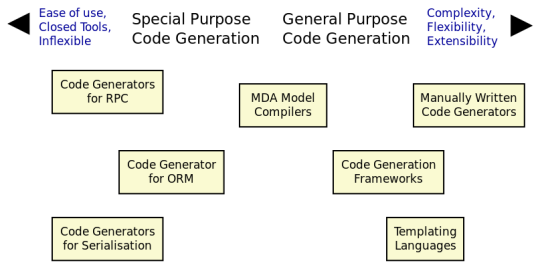

Figure 1: Expressive power of code generation.

3.3 Black Box

Special purpose code generators are command line tools with textual input, and are delivered to users as executables. Whilst they can be extended — particularly those that are Free and Open Source Software (FOSS) — the common use case is as a off-the-shelf black box, where users are not required to peer inside in order to use the tool.

In contrast, general purpose code generators are typically frameworks or libraries — building blocks to be assembled by expert users and tailored for their specific domain in a bespoke and, ideally, iterative manner. They evolve with the practitioner's understanding of the domain.

3.4 Audience

The users of special purpose code generators are software engineers, as they generate one very specific aspect of a larger software system and thus must integrate with traditional development.

On the other hand, MDE users may span a large set of engineering roles — from architects, to analysts to developers — depending on the specifics of a particular application.

3.5 Commonalities

From all that has been stated, it may appear there is a gulf between the role of code generation as understood by MDE and special purpose code generators. Whilst there are differences in objectives, it is important not to lose sight of what they have in common.

Applications of MDE that do not target *full code generation* will ultimately require a degree of integration with "traditional" — *i.e.* non-MDE — software engineering practices, in a fashion very similar to special purpose code generators. Hence, there is value in learning about their approach.

4 Study Method

This section explains the criteria used to select the special purpose code generators, the format of the description for each tool, and the dimensions used for evaluation.

4.1 Selection Criteria

Our criteria for tool selection was as follows:

- **Openness**: FOSS is developed out in the open amongst a community of developers, and thus benefits from a wide range of views. In addition, Open Source projects provide visibility of the health of their development community and development processes, making them the ideal candidates for our research.

[10] As an example of a special purpose code generator that can eschew structural variability, consider a build file generator that needs only a fixed structural input — *i.e.* one or more sets of files.

[11] An idea inspired from Groher and Völter's analysis on the expressive power of DSLs [GV07].

- **Maturity**: The chosen tools must have existed for five years or more and are known to be used in industry. This ensures the approach has been validated and is production ready.

- **Activity**: Projects were required to have been continuously maintained during their lifetime, with a cadence of releases and/or recent commits to their Version Control System (VCS). Both major and minor releases were included in the release count, as per tagging in the project's VCS repository.

- **Diversity**: In the interest of variety, we only selected a project for each given domain in order to obtain better coverage.

From a preliminary list of tools that matched our selection criteria, we selected four tools. The final selection was based on our familiarity with the programming language (C++) and with the tools themselves, in order to facilitate the analysis. It is important to note that the selection is not intended to be exhaustive. Instead, the objective was to survey a small sample set in search of interesting insights. See Section 4.4 for more details on limitations.

4.2 Tool Description

Each surveyed tool has four dedicated sections:

- **Overview**: Brief summary of the generator and its domain, including a summary with items from the selection criteria as outlined in Section 4.1 and a trivial example of the tool's input.

- **Usage**: A walk-through of a typical use of the tool.

- **DSL**: A short description of the DSLs used by the tool, with usage examples where available.

- **Variability Strategy**: A description of the approach to variability taken by the tool.

- **Evaluation**: An evaluation of the tool according to the dimensions defined in the next section.

4.3 Evaluation

The starting point for our evaluation was Whittle *et al.*'s "Taxonomy of MDE Tool Considerations" [Whi+17]. The taxonomy was adapted for the needs of the present study by removing categories and sub-categories which were not deemed applicable, and renaming or merging others for clarity. The final result is the following set of categories:

- **Usability**: General commentary on usability concerns for the tool.

- **Tooling Integration**: How well does the tool integrate with existing development environments and build systems.

- **Code Integration**: How well does the generated code integrate with existing code and build systems.

- **Variability**: Analysis of the trade-offs made between variability and complexity.

- **Dependencies**: Is the generated code self-contained or does it introduce additional dependencies.

- **Generated Code**: Comments on the subjective qualities of the generated code.

- **Error Reporting**: Describes how errors are reported to users.

As with Whittle *et al.*'s taxonomy, its important to note that these categories are not entirely orthogonal — meaning they interact with each other and, in some cases, classification may be ambiguous. However, they are believed to be sufficient for the purposes of the present evaluation.

4.4 Limitations

A survey of this nature is not without its limitations, which must be taken into account in order to ensure applicability. First and foremost, there is a risk in overreaching when using analogies. MDE and special purpose code generators have very different roles in software engineering, leading us to limit our analysis to areas where the overlap is most evident.

Secondly, the focus of the present work was on FOSS as it is more amenable to analysis; however,

proprietary tooling may have a very different set of characteristics due to its development model.

Thirdly, due to familiarity, our focus is skewed towards C++, a compiled language with no reflection support. Given its current focus on performance and systems programming, patterns observed in C++ may not necessarily extend to more modern languages like Java and C# or to interpreted languages.

Fourthly, the chosen sample size was kept deliberately small, mainly in order to allow delving deeper into the functionality of each tool but also because many of the FOSS code generators target similar domains — in particular, cross-language serialisation. Therefore, patterns present in this sample may not be representative of the wider landscape of special purpose code generation, though in our personal experience, we believe they are.

Nevertheless, even taking into account these limitations, we believe the present paper still presents valid suggestions for the development of code generators under MDE. The onus is on the practitioner to ensure applicability and to take into account the listed limitations.

5 Survey

This section introduces all the tools that are part of the survey.

5.1 ODB

ODB[12] is a command line tool that generates Object-Relational mappings for the C++ programming language. It uses suitably annotated C++ source code as its input, and has the ability to generate mappings for a number of Relational Database Management Systems (RDBMSs).

As per the project's website [Syn18b], "[ODB] allows you to persist C++ objects to a relational database without having to deal with tables, columns, or SQL and without manually writing any mapping code." ODB outputs both C++ mapping code and SQL statements to create the relational database schema as well as querying, inserting, deleting or updating mapped entities.

ODB makes use of a set of handcrafted libraries which are referenced by generated code. These provide high-level interfaces for database access, as well as implementations for RDBMS specific functionality.

[12] https://www.codesynthesis.com/products/odb

Table 1: Fact sheet for ODB.

Domain	ORM
First Release	v1.0, September 2010
Latest Release	v2.4, May 2015
Total Releases	20
Latest Commit	May 2018
License	GPL, NCUEL
Input	C++ ODB Pragma Lang.
Output	C++, SQL

Finally, an important aspect of ODB is its implementation as a GCC plugin. Due to this, it is has the same level of compliance with the C++ standard as the compiler, which is very advantageous as the language is very complex and changes frequently.

5.1.1 Usage

ODB is designed to be called as part of the build process in a fashion similar to C++ compilers. It makes very few requirements of the build system, other than the ability to call external programs.

Typically, each invocation of the tool contains one or more target header files which are decorated with ODB pragmas, as exemplified in Listing 13.

```
#include <string>

#pragma db object
class person {
public:
    person() {}

public:
    #pragma db id
    std::string name_;
    unsigned int age_;
};
```

Listing 1: C++ class with ODB pragma annotations.

Users are expected to generate build system rules for each file that requires mappings, as well as rules to compile the generated code into object files. They must also install the ODB supporting libraries, and configure the build system to locate and link the generated code against them.

5.1.2 DSL

ODB defines two internal DSLs, hosted within the C++ programming language. The first is the *ODB*

Pragma Language, as demonstrated in Listing 13. Pragma directives are an extensibility mechanism for the C and C++ languages, and are often used to control implementation specific behaviours of compilers. ODB makes use of it to define ORM related constructs.

According to the manual, the ODB Pragma Language

> [...] is used to communicate various properties of persistent classes to the ODB compiler by means of special `#pragma` directives embedded in the C++ header files. It controls aspects of the object-relational mapping such as names of tables and columns that are used for persistent classes and their members or mapping between C++ types and database types. [Syn18a]

The second Domain Specific Language (DSL) is the ODB Query Language, described as

> [...] an object-oriented database query language that can be used to search for objects matching certain criteria. It is modeled after and is integrated into C++ allowing you to write expressive and safe queries that look and feel like ordinary C++." [Syn18a]

5.1.3 Variability Strategy

ODB offers variability support at two levels:

- **Global**: Invocations of the ODB tool can inline all command line parameters or instead supply an external text file with the configuration. These parameters will affect all applicable entities.
- **Local**: In the source code, each mapped entity can be annotated with pragmas that configure code generation.

When combined, these result in a large configuration surface to control ODB's behaviour. Parameters can be grouped into the following broad categories:

- **Customisation of Relational Entities**: Supply or override names (database name, schema name, index name, table name and so forth), add a prefix or post-fix to relational names, *etc.*
- **Mapping Customisation**: Manually override the default mappings of C++ types to SQL types, or supply a different mapping profile; users can choose a profile that is most suitable for their C++ programming environment — *e.g.* standard C++, Boost or Qt.
- **Customisation of Generated Code**: Add user supplied epilogues and prologues, place generated code in a user-defined namespaces, change the extension and/or names of generated files, configure the version of the C++ standard, the export of symbols, definition of macros, omit the generation of some aspects — *e.g.* do not generate SQL insert statements, queries, *etc.*
- **Database Specific Parameters**: A number of parameters are specific to a given RDBMS, such as the client tool versions, warnings, *etc.*
- **Tracing and Debugging**: Provide debug information of the code generation process, stop generation if the size of generated code is greater than N lines of code, *etc.*

ODB's flexible approach to variability does not preclude a minimalist use case due to its judicious use of default values. The only mandatory parameters are local pragmas in source code to identify entities to map and global command line arguments to point to the target file.

5.1.4 Evaluation

ODB can be characterised across the following dimensions.

- **Usability**: Due to its command line interface mimicking a compiler, ODB has a very shallow learning curve for developers. In addition, by making use of internal DSLs hosted within C++, it requires little learning for a typical C++ developer.
- **Tooling Integration**: By making very few demands of the build system and using C++ source code with few modifications as its input,

ODB is able to integrate with any development environment and build system. Users need not change their setup in order to use ODB.

- **Code Integration**: ODB uses a forward-engineering approach, imposing a strict separation between handcrafted code and generated code. Generated code is not intended to be modified by its users; changes must be exclusively made to the handcrafted source code via the ODB Pragma Language followed by regeneration.
- **Variability**: ODB supports a high-degree of variability but requires very little configuration in order to produce code. This lowers the barrier of entry to new users.
- **Dependencies**: Generated code requires ODB specific libraries. Whilst producing smaller and simpler code, this also means having to install the libraries and configure the build system to find them, as well as adding dependencies to the deployment.
- **Generated Code**: Samples of the generated code produced by ODB were manually inspected and found to be of a standard comparable to the handcrafted code of the ODB libraries. This is very advantageous when debugging and troubleshooting problems. In addition, ODB offers a number of options dedicated to customisation of generated code, easing the integration into existing code bases.
- **Error Reporting**: Error messages are reported to the command line using the formatting defined by the GCC compiler. This is less convenient for users of other compilers — such as Microsoft Visual C++ — as their development environment may not able to interpret error messages.

5.2 Protocol Buffers

Protocol Buffers are a cross-platform serialisation mechanism for structured data, allowing the exchange of messages in possibly heterogeneous environments such as different hardware platforms and programming languages. Protocol Buffers has four main components: a language for the definition of messages, a so-called "compiler" that transforms the message definition into source code, a wire-format that specifies its binary representation and helper libraries that are referenced by the generated code.

Table 2: Fact sheet for Protocol Buffers.

Domain	Structured data serialisation
First Release	v2.0, July 2008
Latest Release	v3.5 November 2017
Total Releases	19
Latest Commit	June 2018
License	BSD
Input	Protocol Buffers Language
Output	Multiple languages

Whilst there are multiple implementations available, our survey focuses on the default protocol buffer compiler `protoc` as supplied by the Protocol Buffers project. The compiler has out of the box support for several programming languages such as C++, C# and Java.

In addition to code generation, `protoc` also has the ability to encode and decode messages, but, this functionality is out of the scope of the present analysis.

5.2.1 Usage

Users define one or more structured data types in a text file, written in conformance with the Protocol Buffers Language [Goo18b]. Input files typically have the extension `.proto`. Listing 7 provides an example message.

```
syntax = "proto3";

message person {
  string name = 1;
  int32 age = 2;
}
```

Listing 2: Message using Protocol Buffers IDL.

Each invocation of the tool is made against one or more `.proto` files and must supply command line parameters to determine the set of programming languages to generate. Other than the ability of calling external binaries, the compiler makes very few demands from the build system — thus supporting all modern build systems.

Users are responsible for creating build system rules to transform the `.proto` files, as well as rules to compile the generated code into object files as required by the target language. However, the

tool supports the automated generation of rules for `make`-like build systems. Finally, generated code depends on handcrafted libraries supplied by the Protocol Buffers project, so these must be installed and made visible to the build system.

5.2.2 DSL

As described previously, `.proto` files must conform to the Protocol Buffers Language [Goo18b], currently at version 3. The language has a C-like syntax, and provides a set of constructs from the domain of message serialisation such as:

- Message definition;
- Ordering of fields in a message;
- Optional, mandatory and reserved fields;
- Primitive types with well-specified machine-level representation, independent of target platform.

The parsing and validation of `.proto` files is performed by `protoc` as part of the generation process.

5.2.3 Variability Strategy

Outside of the structural variability enabled by the creative construction nature of the Protocol Buffers Language, `protoc` has very limited support for variability. All of its parameters are global, and fall under the following categories:

- **Output**: Determines if an output language is enabled, the location for its files, whether to concatenate output files, user-created plugins to add support for additional programming languages, *etc.*
- **Tracing and Debugging**: Format of error messages, list available free fields, *etc.*

5.2.4 Evaluation

The `protoc` compiler can be characterised across the following dimensions.

- **Usability**: The Protocol Buffers DSL is very similar to typical programming language constructs and other Interface Description Languages (IDLs) such as CORBA, which greatly facilitates learning. The Protocol Buffers compiler has a very simple command line interface, allowing users to generate code with minimal knowledge of the infrastructure.
- **Tooling Integration**: The compiler is designed to fit in the existing build systems and development environments, needing very little support in order to do so. It also behaves in a fashion similar to other development tools such as linkers and compilers, facilitating integration.
- **Code Integration**: `protoc` uses a forward-engineering approach, so users are not allowed to modify generated code. The generated code is expected to be integrated with the remaining code for the system via rules in the build system.
- **Variability**: The constrained variability approach taken by `protoc` reduces the learning curve, but as a consequence it is not possible to customise generated code to handle specific use cases such as adding epilogues or prologues, changing namespaces, etc.
- **Dependencies**: Generated code requires Protocol Buffers specific libraries. These help keep generated code small, but demand additional setup from the build system in terms of locating dependencies and additional artefacts to deploy.
- **Generated Code**: Upon inspection, we found that the quality of the generated code for C++ is not at the same level as the handcrafted code in supporting libraries. Nevertheless, the code is simple enough to enable users to debug it.
- **Error Reporting**: `protoc` provides the ability to report errors using either GCC or Microsoft's Visual Studio formats, thus integrating with two of the major development environments for C++. However, other programming languages may have different notations for the reporting of errors, and thus do not benefit from the same level of integration.

5.3 SWIG

Simplified Wrapper and Interface Generator (SWIG)[13] reads C and C++ code and generates the infrastructure necessary to allow calling the original code from a different programming language. SWIG originally targeted scripting languages but over time it has been extended to support compiled languages as well such as Java and C#.

Table 3: Fact sheet for SWIG.

Domain	Language interoperability
First Release	v1.0 September 1996
Latest Release	v3.0.12 January 2017
Total Releases	68
Latest Commit	June 2018
License	GPL
Input	C/C++, SWIG interface
Output	Multiple languages

The SWIG website states that

> [i]t works by taking the declarations found in C/C++ header files and using them to generate the wrapper code that scripting languages need to access the underlying C/C++ code. In addition, SWIG provides a variety of customization features that let you tailor the wrapping process to suit your application. [Pro18]

An important area where SWIG has limitations is in the parsing of C++ code, as it uses an internal C/C++ parser. Due to the complexity of the C++ language, as well as its fast pace of change, the parser is not able to parse all compliant C++ code — particularly code that makes use of features in the latest standards, *e.g.* C++ 14, C++ 17.

5.3.1 Usage

Whilst SWIG is able to parse C and C++ code directly, the recommended usage is to create a separate SWIG interface file that explicitly defines the Application Programming Interface (API) to export. This is done so as to avoid exporting types inadvertently and also to stop polluting general source code with SWIG annotations. Interface files typically have a `.i` or `.swg` extension and contain C/C++ code interspersed with SWIG interface commands, as exemplified in Listing 14.

```
%module people
%{
#include <string>
%}

class person {
public:
    person() {}

public:
    std::string name_;
    unsigned int age_;
};
```

Listing 3: C++ class with SWIG macros.

Once defined, the interface files can be processed by the command line tool `swig`. Users can choose to generate wrappers for one or more languages by supplying command line arguments.

SWIG does not make any demands on the build tool, other than the ability to call external processes, so it integrates with most build systems. It is the responsibility of the user to create appropriate build system rules to generate the wrapper code and to build the shared objects that ultimately will be used in the target language.

5.3.2 DSL

The DSL used by SWIG in its interface files is based on the C pre-processor, itself a simple text processing language. The SWIG pre-processor adds its own set of commands, escaped with `%`. The main objective of the SWIG commands is to allow a fine grained control over the exported API.

The following is a sample of the available commands:

- `%include`: Includes a file into the interface. The original pre-processor `#include` is ignored to avoid including files into the API unnecessarily such as library headers and other third party code.
- `%import`: Includes a file to satisfy dependencies, but does not add its contents to the exported interface.
- `%define, %inline, %enddef`: Provides a more convenient interface for macro definition at the SWIG level.
- `%extend`: Extends an existing class interface with additional code.

[13] http://www.swig.org

- `%typemap`: Provides a way to override the default mapping of types.
- `%module`: Defines a containing module for the exported code. The notion of "module" is mapped to the adequate construct in the target language such as `namespace` in C# and `package` in Java.

The pre-processor commands have evolved over the years to cater for a large range of use cases in interoperability, and thus addresses the majority of requirements.

5.3.3 Variability Strategy

The SWIG DSL produces transformations on the original C and C++ source code, and thus it is a creative construction DSL focused on structural variability.

The remaining support for variability in the `swig` tool is very limited, and falls under the following categories:

- **Input**: Add support for C++ (only C is supported by default), change the behaviour of the C pre-processor, *etc.*
- **Output**: Configuration of the languages to generate, directories in which to output the files, *etc.*
- **Tracing and Debugging**: Dump information on the API to generate, dump symbol tables, dump type mapping, show code after pre-processing, set the warning level, *etc.*

5.3.4 Evaluation

SWIG can be characterised across the following dimensions.

- **Usability**: The `swig` tool itself requires a shallow learning curve, since it uses a command line interface similar to that of a compiler and has a small the number of configuration options — most of which are common to a compiler. However, the SWIG DSL does not share these properties. SWIG interface files — with its two-stage pre-processing pipeline and two sets of pre-processing commands — can become very large and complex and require developers that are knowledgeable about SWIG.
- **Tooling Integration**: The `swig` tool is designed to fit in the existing build systems and development environments by following a workflow similar to a compiler.
- **Code Integration**: SWIG uses a forward engineering approach, thus generated code is not modifiable. Users are expected to design build system rules to build and link the generated code in the same manner as for other handcrafted code.
- **Variability**: On one hand, the structural variability promoted by the SWIG DSL makes the tool highly configurable and able to handle a variety of very complex use cases. On the other hand, outside of simple scenarios, SWIG has a very steep learning curve due to this support for variability.
- **Dependencies**: Generated code does not have any third-party dependencies, which makes it easier to integrate.
- **Generated Code**: SWIG generates thousands of lines of C++ code even for trivial examples, making it difficult to understand. The authors of SWIG state this clearly in the generated code via the following comment: "This file is not intended to be easily readable and contains a number of coding conventions designed to improve portability and efficiency." Unsurprisingly, the quality of generated code is lower than handcrafted code, but it is well-structured and simple enough to make debugging possible.
- **Error Reporting**: The `swig` tool reports errors using the GCC output formatting, which makes integration with environments using GCC straightforward. However, it does not support Microsoft's Visual Studio format.

5.4 XSD

XSD[14] is a tool that receives an XML schema[15] as input and outputs C++ classes representing the entities in the schema, as well as XML serialisation code for those classes.

As per the project's website,

> the biggest advantage of this approach is that you can "[...] access the data stored in XML using types and functions that semantically correspond to your application domain rather that dealing with the intricacies of reading and writing XML. [Syn18c]

Table 4: Fact sheet for the XSD tool.

Domain	XML mapping
First Release	v1.0, August 2005
Latest Release	v4.0, September 2014
Total Releases	19
Latest Commit	November 2017
License	GPL, NCUEL (proprietary)
Input	XML Schema
Output	C++

XSD provides two backends for the generated code: *parser* and *tree*. The parser backend uses streaming for document processing, which is more suitable when handling large documents, or for simpler access patterns. The tree backend loads documents in its entirety to an in-memory tree, and is designed for for smaller documents and more complex access patterns. Backends are selectable via command line options.

5.4.1 Usage

Users create XML schemas using their XML editing tool of choice. Once defined, the XML schema is supplied to the command line tool `xsd-4`.[16] Listing 13 provides an example XML schema that can be used as input to XSD.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs=
 "http://www.w3.org/2001/XMLSchema">
 <xs:complexType name="person">
  <xs:attribute
      name="name"
      type="xs:string"/>
  <xs:attribute
      name="age"
      type="xs:integer"/>
 </xs:complexType>
</xs:schema>
```

Listing 4: XML Schema input for XSD tool.

The command line tool generates the C++ classes and the XML mapping code; the user must then integrate the generated code into the build system by creating the required build system rules. However, if the build system is a variant of `make`, the tool can also be used to code generate the rules.

In addition, the generated code depends on handcrafted libraries, so the onus is on the user to install these and to make them visible to the build system.

5.4.2 DSL

XML is a mature, standardised language for describing structured data [W3C08]. There are a variety of tools for editing and processing XML documents and schemas. Due to this, the DSL is completely decoupled from the XSD tool.

5.4.3 Variability Strategy

In addition to the structural variability enabled by XML schemas, the XSD tool has a number of parameters to configure the generation of code. These can be classified into the following broad categories:

- **Backend**: As discussed above, the type of XML processing to generate code for. Some of the options are only applicable to a specific backend.
- **Output**: Directory in which to place the output, whether to generate one file per type, *etc.*
- **Mapping Customisation**: Override the default type mapping between C++ types and XML types.
- **Customisation of Generated Code**: Which C++ standard to target, what character encoding to use, whether to inline functions, override filenames and extensions, modify include paths with regular expressions, header guards, *etc.*

[14] https://www.codesynthesis.com/products/xsd

[15] XML schemas are also known as XSDs, giving the name to the tool. However, in the interest of clarity, we will only refer to them as *XML schemas* in this paper.

[16] The tool name may vary depending on your installation.

- **Build System**: Options related to the generation of build system targets for generated code.
- **Tracing and Debugging**: Limit generation to a given number of lines of code, tracing of regular expressions, *etc.*

Whilst there are a large number of command line parameters, the XSD tool is able to generate code with very little configuration supplied, due to the use of defaults. The only mandatory parameter is the backend.

5.4.4 Evaluation

The XSD tool can be characterised across the following dimensions.

- **Usability**: The command line interface provided by the tool is similar to other command line tools such as compilers, thus lowering the learning curve for new users. In addition, users can use their XML editing tool of choice to create and validate the input XML schema.
- **Tooling Integration**: The tool integrates with any modern build system and development environment that supports calling external tools.
- **Code Integration**: The XSD tool uses a forward engineering approach, meaning that generated code should not be modified and is separated from handcrafted code. Changes are made to the XML schema and code is regenerated.
- **Variability**: Outside of the creative construction of XML schemas, the XSD tool supports a high-degree of variability which enables users to customise the generated code for their particular use case. However, due to defaulting, the tool requires very little customisation in order to generate code, resulting in a low barrier of entry for new users.
- **Dependencies**: By requiring the installation of dependencies in order to build the generated code, the XSD tool made the setup process more complex than it would have otherwise been without dependencies.
- **Generated Code**: The code generated by the XSD tool is of a standard comparable to the handcrafted code in their core libraries. It is well-commented and succinct, largely due to its reliance on external libraries.
- **Error Reporting**: Errors are reported using the GCC formatting, enabling an easy integration to environments which use this compiler. However, given that the input is in standard XML, users can ensure the document is valid via their XML tool of choice before code generation.

6 Best Practices and Lessons Learned

Whilst covering four different domains, all the four tools under analysis nevertheless presented a number of commonalities, which can be broadly categorised under the following themes.

6.1 Low Barrier to Entry

In all four cases, the tooling presented a low barrier to entry to new users by trying to keep complexity low. This results emerges from a number of decisions:

- **Simple Workflow**: All tools under analysis had a very similar workflow, roughly mimicking a typical C++ compiler. The workflow is kept simple due to a reliance on forward-engineering, therefore bypassing complex integration issues with handcrafted code.[17]
- **Ease of Integration**: The surveyed tools make very little demands in terms of integration in a wider environment, and thus can be used by any build system. The ease of integration extends to error reporting, though here most tools only supported the GCC error formatting. In addition, users are not required to change their project structure in order to cater for generated code, which further eases integration. In some cases its even possible to make generated code look like handcrafted

[17] For a good description of integration issues see Greifenberg *et al.* [Gre+15].

code by adding epilogues and prologues — useful for comments, licences and related boilerplate.

- **Ease of Use**: All tools can be used with a very small number of mandatory configuration parameters, making it easy to get started. In general, tools required minimal understanding of the underlying domain of the tool and no experience with the modeling and code generation domains.

- **Ease of Troubleshooting**: Whilst the quality of generated code varied from tool to tool, in general all of them produced code that is suitable for debugging. A subset of the tools produced high-quality code, comparable to handcrafted code.

6.2 Input Decoupling

All surveyed tools have a command line interface with textual input, which is a characteristic of special purpose code generators. This architecture has a number of advantages:

- The audience of the tool becomes more focused. The code generator can be specifically designed for software engineers without any need to accommodate other types of users.

- Users need not change their development environment to manage the input files — though, if the input is a DSL specific to the tool, there may be a need for additional plugins in order to obtain a rich editing environment.

- Input files can reuse the same VCS as source code, the same process of code reviews, *etc.* From a process perspective, they are treated like ordinary source code.

- If the input is defined by an external specification such as XML, users benefit from the existing tooling ecosystem.

- Textual input does not preclude graphical editing; external tools can provide graphical manipulation, as long as they are able to generate the textual input. However, this is not a code generator concern.

- Input files can be pre-processed by other tools, including pipelines via the chaining of tools. For example, post-processing scripts can be applied for pretty-printing, filtering, *etc.* As before, these are not a code generator concern, which makes the process flexible.

Most of these items are corollaries of the Unix Philosophy, which McIlroy succintly describes as follows:

> Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface. [Sal94]

6.3 Incremental Complexity

Most tools require very little theoretical understanding when getting started. A subset of the surveyed tools have a large variability surface that can be deployed by advanced users to handle specific use cases, but which is hidden from beginners and intermediate users via defaulting mechanisms. Users can explore the surface incrementally, as they become proficient with the tool. However, in some cases such as SWIG, advanced use cases result in very complex input files.

7 Conclusion

The present study performed a survey of four special purpose code generators across four distinct domains, and extracted a set of best practices and lessons learned from their approach. The conclusions of this work take the form of recommendations, which should be considered only when not making use of *full code generation*.

The recommendations are as follows:

- MDE tools should decouple the modeling functionality from code generation, and consider using a textual DSL to communicate between modeling and code generation.

- The code generator should be a command line tool with an interface similar to that of a compiler — including error reporting — in order to integrate seamlessly with most build tools

and to better focus on its audience — software engineers.

With regards to future work, an interesting direction of research may be to perform a broader but shallower survey, spanning across a large number of special purpose code generators, with several tools per domain and covering multiple programming languages. In addition, the categories used for tool evaluation may also provide material for an extension of Whittle *et al.*'s "Taxonomy of MDE Tool Considerations" [Whi+17].

References

[And+14] Luigi Andolfato et al. "Experiences in Applying Model Driven Engineering to the Telescope and Instrument Control System Domain". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2014, pp. 403–419 (cit. on p. 2).

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Vol. 1. 1. Morgan & Claypool Publishers, 2012, pp. 1–182 (cit. on pp. 1, 3).

[CH03] Krzysztof Czarnecki and Simon Helsen. "Classification of model transformation approaches". In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. Vol. 45. 3. USA. 2003, pp. 1–17 (cit. on p. 3).

[CH06] Krzysztof Czarnecki and Simon Helsen. "Feature-based survey of model transformation approaches". In: *IBM Systems Journal* 45.3 (2006), pp. 621–645 (cit. on pp. 1, 3).

[EA+15] Yosser El Ahmar et al. "Enhancing the communication value of UML models with graphical layers". In: *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*. IEEE. 2015, pp. 64–69 (cit. on p. 1).

[GV07] Iris Groher and Markus Voelter. "Expressing feature-based variability in structural models". In: *In Workshop on Managing Variability for Software Product Lines*. Citeseer. 2007 (cit. on pp. 3, 4).

[GV09] Iris Groher and Markus Voelter. "Aspect-oriented model-driven software product line engineering". In: *Transactions on aspect-oriented software development VI*. Springer, 2009, pp. 111–152 (cit. on p. 4).

[Goo18a] Google. *Protocol Buffer Basics: Java*. 2018. URL: https://developers.google.com/protocol-buffers/docs/javatutorial (visited on 05/22/2018) (cit. on p. 3).

[Goo18b] Google. *Protocol Buffers Language Guide (proto3)*. 2018. URL: https://developers.google.com/protocol-buffers/docs/proto3 (visited on 05/22/2018) (cit. on pp. 8, 9).

[Gre+15] Timo Greifenberg et al. "Integration of handwritten and generated object-oriented code". In: *International Conference on Model-Driven Engineering and Software Development*. Springer. 2015, pp. 112–132 (cit. on p. 13).

[Hut+11] John Hutchinson et al. "Empirical assessment of MDE in industry". In: *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE. 2011, pp. 471–480 (cit. on p. 1).

[JMS08] Sven Jörges, Tiziana Margaria, and Bernhard Steffen. "Genesys: service-oriented construction of property conform code generators". In: *Innovations in Systems and Software Engineering* 4.4 (2008), pp. 361–384 (cit. on p. 1).

[Mus+14] Gunter Mussbacher et al. "The relevance of model-driven engineering thirty years from now". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2014, pp. 183–200 (cit. on pp. 1, 2).

[PV12] Richard F Paige and Dániel Varró. "Lessons learned from building model-driven development tools". In: *Software & Systems Modeling* 11.4 (2012), pp. 527–539 (cit. on p. 2).

[Par85] David Lorge Parnas. "Software aspects of strategic defense systems". In: *Communications of the ACM* 28.12 (1985), pp. 1326–1335 (cit. on p. 2).

[Pro18] The SWIG Project. *SWIG - Executive Summary*. 2018. URL: http://www.swig.org/exec.html (visited on 06/06/2018) (cit. on p. 10).

[Ros+12] Louis M Rose et al. "A feature model for model-to-text transformation languages". In: *Modeling in Software Engineering (MISE), 2012 ICSE Workshop on*. IEEE. 2012, pp. 57–63 (cit. on p. 3).

[SKSG07] Dov Shirtz, Michael Kazakov, and Yael Shaham-Gafni. "Adopting model driven development in a large financial organization". In: *European Conference on Model Driven Architecture-Foundations and Applications*. Springer. 2007, pp. 172–183 (cit. on p. 2).

[Sal94] Peter H Salus. *A quarter century of UNIX*. Addison-Wesley Reading, MA, 1994 (cit. on p. 14).

[Stö14] Harald Störrle. "On the impact of layout quality to understanding UML diagrams: size matters". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2014, pp. 518–534 (cit. on p. 1).

[Syn18a] Code Synthesis. *C++ Object Persistence with ODB*. 2018. URL: https://www.codesynthesis.com/products/odb/doc/manual.xhtml (visited on 06/04/2018) (cit. on p. 7).

[Syn18b] Code Synthesis. *ODB Website*. 2018. URL: https://www.codesynthesis.com/products/odb (visited on 06/08/2018) (cit. on p. 6).

[Syn18c] Code Synthesis. *XSD Website*. 2018. URL: https://www.codesynthesis.com/products/xsd (visited on 06/08/2018) (cit. on p. 12).

[Völ09] Markus Völter. "MD* Best Practices". In: *Journal of Object Technology* 8 (2009), pp. 79–102 (cit. on pp. 1, 3).

[Völ+13] Markus Völter et al. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013 (cit. on pp. 1, 3).

[W3C08] W3C. *Extensible Markup Language (XML)*. 2008. URL: https://www.w3.org/TR/xml/ (visited on 06/06/2018) (cit. on p. 12).

[Whi+17] Jon Whittle et al. "A taxonomy of tool-related issues affecting the adoption of model-driven engineering". In: *Software & Systems Modeling* 16.2 (2017), pp. 313–331 (cit. on pp. 1, 2, 5, 15).