Code Generation & Meta-Programming

13 March 2017 | **Insight Zuhlke, Software Engineering** | **Edgar Holleis**
Reading time: 10 minutes

Code generation and meta-programming can be significant time savers in embedded software projects. But project leaders and developers are reluctant to use it, for various reasons. This post looks into these reasons and analyses them one at a time. In conclusion, only some of the reservations towards code generation tools seem to be justified.

By code generation I mean tools that output C code intended to run on the embedded system or the wider context (test system, support code). There are some problems that typically lend themselves to code generation:

- Parsers (text parser, binary parsers)
- Finite-state machines (FSM)
- Digital filters (e.g. signal processing)
- I/O routing on modern microcontrollers

There are two things all these problems have in common: First, they cannot be elegantly solved by using a traditional library. Secondly, there is some kind of model involved. The model specification serves as input to a code generation tool and the tool computes a solution that is often more efficient and contains less defects than a hand-coded solution. Added benefit, the model specification doubles as documentation for the problem being solved.

Code generation tools save time and effort because developers can concentrate on specifying the problem and let the tool deal with the implementation. The alternative is implementing by hand, and relying on informal specification (i.e. a Word document) only. This is dangerous for several reasons. First – if the specification changes at a later point, the hand-coded implementation will need to be reworked by hand. And even more importantly – for any problem that can be described by a formal model, it is probably a good idea to use the model and stick to its formalism. Even better is to follow the model rigorously. A laissez-faire approach to model compliance is a risky endeavour. An uncleanly implemented parser may have serious security implications. Sloppy state machines may have serious safety repercussions. The best way to guarantee correctness with regards to a model is to rely on formal specification and a tool to generate the implementation automatically.

More than just state machine generators

Examples for code generation tools are numerous. There are the venerable “lex” and “yacc” for generating parsers, or “antlr”, a contemporary solution. For FSMs there are numerous lightweight tools to big model driven development packages. A commercial tool geared at embedded development is SinelaboreRT. There are digital filter design tools of all sorts (academic, electronic design packages, Matlab, …). But a code generation tool can be as benign as an Excel Sheet, a Python script or an in-house developed C# application.

Meta-programming is another useful approach that comes to mind. For the sake of argument, I would consider meta-programming as a special case of code generation, not an alternative. Instead of using a dedicated tool, some intrinsic mechanism of the host language is used, such as the C++ template engine or the C pre-processor. Just like a dedicated code generation tool, a template-library or meta-program may be viewed as a program in its own right. What "expanding" a macro or "instantiating" a template really means is that the compiler of the host language takes the template parameters as inputs, interprets template code and generates concrete (i.e. non-meta) code as output. Also, with any non-trivial template library, there is often a model involved and the template instantiation can be viewed as sorts of model specification. For examples look no further than the C++ Boost libraries ("msm" state machines, "spirit" parsers).
On to the core issue of this post: Of course there are disadvantages to code generation and arguments that people bring forward. But only some of them are justified. More specifically, we will analyse the following three counter arguments:

1. Code generation tools are complicated.
2. Code generation tools need to be certified, its output validated and tested.
3. Code generation tools complicate the build.

Let's look at these one by one:

Counter argument: Tools are complicated

The first argument has to do with skills: Managers worry that using code generation tools increases the costs of training, of bringing fresh developers up to speed: It's hard enough to find decent C programmers. It is all but impossible to find developers who happen to know just the right combination of code generation tools.

In my opinion that kind of thinking misses the big picture: Parsers, finite state machines and filters all have a rich and extensive theoretical background. What you really need are developers who have a certain understanding of this theoretical background. In that sense, the code generation tools actually lower the required developer qualification for project success. This is because code generation tools themselves embody experience about the underlying theory. Using the proper tool, somebody with only basic understanding can arrive at a decent solution. On the other hand, a developer who is unable to use a parser generator (or FSM generator or filter design toolbox), should not be trusted with hand-coding a parser (or FSM or filter).

10 years from now, there still will be parsers (and FSMs and filters). The theoretical foundations may have slightly evolved, but will not have fundamentally changed. There may be new tools. Or not. Anyway, with a robust model and a formal specification you will be in a much better position to restart an old project, than with a hand-coded mess (yes, by the time it will have evolved into a mess) and no specification.

Counter argument: Tools need to be certified

The second argument is about the need to get the tool certified and validate its output. That is no less the case than for any library, compiler or third party component. And as with any tool, the certification work can be amortized over several projects.

Other than the code generation tool itself, the question remains whether the tool's input and output must be tested and certified. It is a tricky question and can easily get philosophical. Is a tool's input, i.e. the model specification, rather like source code or rather like a specification document? The truth lies somewhere in between. The questions that need to be asked are whether the model specification solves the right problem and whether it solves the problem correctly.

Let's look at a concrete example – a parser. The questions whether a parser correctly parses the intended inputs and correctly rejects non-complying inputs can be answered with relative ease by looking at its specification – typically some form of EBNF (a specification language for parsers). Answering those questions by looking at a hand-coded implementation of a parser is a major headache. The formalism embedded in the parser generator makes validation easier, not harder. A similar argument can be made about FSMs and digital filters.

Does a code generation tool's output need to be validated? The answer is "probably not". This is because the tool itself will already have been validated and therefore can be trusted to correctly transform the model specification into correct source code. There is little added value in unit-testing or formally reviewing the generated source code. However, the generated source code is embedded in some larger component and you will definitely want to test correct behaviour of this larger component. It may be viewed as integration test between handwritten source code and generated source code.

This approach to testing may be at odds with requirements like "100%" decision coverage. However, formalisms like FSMs come with their own test theory and provide their own metrics, e.g. "state transition coverage". Digital filters can be tested with appropriate test signals and can be analysed for numerical stability. Application of domain specific test theories can thus inspire more confidence in the system than generic source code metrics. I would rather trust a filter that passes carefully selected test signals and is proven numerically stable, than a filter that is unit tested for 100% decision coverage. I therefore suggest not applying generic source code metrics to generated code. Instead, the model behind the code generation tool is the key for devising an effective testing and validation strategy. Maybe the tests can even be partly auto-generated based on the model specification.

What if the model specification contains snippets of source code that end up in the generated output? This is the case for many popular code generation tools, including classics like “lex” and “yacc”. There are two pieces of advice: Keep the code snippets as simple as possible (no branches) and write integration tests that cover each snippet at least once.

Counter argument: Tools complicate the build

The third argument is that code generation tools complicate the tooling. This is especially true if the primary build system is IDE-based. But the trend towards continuous integration somewhat alleviates the overhead caused by code generation tools. Continuous integration means there already is a fully automated infrastructure in place that orchestrates source control tool, compiler toolchain and packaging. A code generation tool is just another step. It is therefore advantageous to use code generation tools that can be automated from the command line and that can thus be integrated into a continuous delivery infrastructure.

The meta-programming approach has the advantage of not requiring an external tool that complicates the build. That needs to be weighed against the fact that meta-programming can be a chore. Even small problems can lead to very involved meta-programs. But most of the time, both methods are better than hand-coding.

Guerrilla code generators

For projects that cannot adopt code generation as part of the official tooling, there is the option of using “passive” code generation. This means generating an initial version and then maintaining the generated code by hand. This works best with code generators that work like template engines – GSL may serve as example. It is a viable approach for cases where you would otherwise have to hand-code a very regular and repetitive problem (e.g. binary encoders / decoders) or where there is a lot of boilerplate involved (e.g. remote procedure calls).I still argue, however, that from the point of view of long-term maintainability, it would be better to adopt the code generator as part of the tooling, view the code generator’s inputs as canonical source form and treat the code generator’s outputs as intermediary results. The input is more abstract, more expressive and closer to the specification. The inputs may be used to generate additional artefacts, such as test cases and documentation. And if the code generation tool proved to be useful during project inception, it will still be useful at later stages, for example when the specification change and you need to rework the system.

Final verdict

There is some reluctance to adopt code generation tools. However, as I tried to show, the arguments against tool usage are misinformed. Yes, tools may be hard to use, but hand-coding is harder. Especially so, if done by developers without the required theoretical background. And yes, tools and the code they generate have to be tested and certified. But it can be done more intelligently than by using unspecific source code metrics and will result in a much more reliable system. And yes, code generation tools do complicate the tooling, but the advantages of adopting tools outweigh the disadvantages.