Software and Systems Modeling (2024) 23:267–268
https://doi.org/10.1007/s10270-024-01172-2

EDITORIAL

Model-based code generation works: But how far does it go?—on the role of the generator

Benoit Combemale[1] · Jeff Gray[2] · Bernhard Rumpe[3]

Published online: 8 April 2024

There are many published examples of successful industrial projects that used code generation from abstract models. However, it seems that the actual use of code generation from explicitly defined models written in modeling languages (e.g., UML, SysML, or a domain-specific language) has not been as successful and widespread as possible.

Why is that so? Unfortunately, there are too few in-depth examinations of the problems and challenges that actually prohibit the widespread adoption of model-based code generation. It would be very instructive for the SoSyM community to have deeper insights into these challenges so that we could improve the current state-of-the-art.

However, there are a number of experiences commonly used as counterarguments. Some of them are:

1. The developers' ability to model in abstract form is not as widespread as it should be. In particular, for developers, it may be easier to think in a linear procedural way, than in an abstract specification.
2. The tooling is not as comfortable and robust as it should be, in particular when developers rely on graphical representations of diagrams that have to be arranged into new layout regularly when changes occur.
3. The generation of code has to react flexibly to the input model, framework APIs, operating system, and other elements of the software stack, which are regularly updated and thus enforce the generator's adaptations.
4. The benefit of modeling is limited if tool support for defining the abstraction levels in a model is weak.

✉ Bernhard Rumpe
bernhard.rumpe@sosym.org

Benoit Combemale
benoit.combemale@sosym.org

Jeff Gray
jeff.gray@sosym.org

[1] University of Rennes, Rennes, France

[2] University of Alabama, Tuscaloosa, AL, USA

[3] RWTH Aachen University, Aachen, Germany

Although these arguments are related, argument (1) is a teaching problem and (2) is a tool vendor issue. Argument (3) obviously needs to be mitigated by a flexible modular approach when defining or adapting a generator. That means not only the models and the resulting code should be organized in a modular way but also the generator. Furthermore, the generator should be easily and flexibly adaptable. Argument (4) is an interesting one and comes from dumb code generators. For example, given a class diagram with a class "Person" and an attribute "age", the diagram is directly mapped to a corresponding implementation class plus the attribute. A minimal extension would be access control to attributes through getter and setter methods. But we can do better!

1 What if the generator was smart?

There would be improved value if a generator could also create a factory or builder, the data table definition for the database, and the DAO to load and store objects. Additionally, it would be helpful to have support for web pages for visualizing or editing single objects, creating new objects, and for lists of available objects. Model-based smart generators can generate data transportation code in different languages (e.g., allowing a Typescript webapp or a Python data crunching algorithm to communicate with a Java backend in a secured way) and could also govern the access, privacy, and transaction policies.

A smart generator knows that an attribute called "temperature" has a different visualization than an attribute called "interestRate". Such domain-specific intelligence in a generator must be extensible for various new domain-specific solutions, e.g., in the finance, retail, research, manufacturing, medicine, government, or e-home domains, where individual forms of presentations have been created already before digitalization started.

A smart generator might also automatically include a log to understand recent changes in the data or a difference mechanism when users would like to know what happened since the last day/week/month. It might automatically add an appropriate undo/redo mechanism. And a smart generator automatically creates the user interface for a representation of these logs, an offering of filters on these logs, undo/redo-buttons, an activity report, an email notification mechanism, etc.

And additionally it could generate sets of possible test data as well as dummies, respectively, mocks for tests. It could also generate automatically specific search functionality including the appropriate webpages showing the search results. And maybe the search is itself smart, knowing how to handle specific forms of data types, such as temperatures or interest rates?

It could generate APIs for secure REST and MQTT access itself based on a generated JSON or XML dialect on the server side to communicate with other systems. It could organize the overall system in forms of micro-services and many more possible architectural issues.

Of course, for data-intensive or highly optimized algorithmic-based applications, the generated code is not necessarily fit enough to meet all non-functional requirements. That either would mean that the generator must be adaptable (for one more reason than only described above) or the generated code must be adaptable, i.e., the generated code should be organized as a framework itself, where unused things can be left out and many other things can be adapted in various forms.

We do not have that currently—although it seems feasible. One important side condition would be that it may not be a 1-shot only generation, but must be repeatable in an agile iterative development process, where the generated code actually is not touched at all but only programmed against—like in a framework. And like in a framework many default design decisions are embedded in the generation tool and in the runtime-part of the generator, which reduces the design time, but might also narrow usage of a generator to certain domains or kinds of software.

Class diagrams have a special role in modeling languages, because they define the structural backbone of a system (and SysML IBDs/BDDs do that in a logically distributed or component-based setting). All other kinds of modeling languages, such as StateCharts or activity diagrams, are typically based on that backbone. But from here more interesting code can be generated. This includes the state pattern, but also visualizations in the web presentation, user notifications (e.g., if certain states have been reached), logs, metrics on the durations of actions, etc. Behavioral descriptions can generally be understood as special forms of descriptions of code behavior, but also as process descriptions to guide the interaction between the software and the users, which leads to a lot of generated user-friendly code.

Even logic formulas, such as OCL, which is relatively close to first-order logic, can be mapped into code. The straightforward mapping is to check an OCL condition, a smarter mapping is to generate test data that exercise several main and corner cases to cover a condition, and a smartest one is to generate code that establishes a post condition. However, such a generator needs a kind of smartness which we know from formal methods and not only smart knowledge about how to represent certain classes, associations and attributes in a user-friendly way. Theoretical computer science has delivered such smartness in the form of solvers for logical constraints, search strategies, and transformations of high-level specifications to efficient implementations for many cases, even though the principal limitation of undecidability will always exist. Currently, it seems that not so many of these techniques are actually in use, which is a pity. Maybe a combination of AI techniques for selecting the appropriate algorithms and these algorithms themselves will lower the usage barriers.

In summary, a smart generator can actually prevent us from a lot of coding, allowing developers to become quicker in developing and smarter in evolving our software. A smart generator uses smart generation techniques for adaptable framework-like code and ships with an enhanced runtime-framework.

There is still much to do, especially for the tool developers who are actually in charge of building industrial strength tooling that is more than just “drawing” or “sketching” tools. Let’s help to adopt the available techniques in industry and identify the mechanisms needed to do so.

Funding Open Access funding enabled and organized by Projekt DEAL.