Building Model-Based Code Generators for Lower Development Costs and Higher Reuse

Hans-Werner Sehring
Department of Computer Science
NORDAKADEMIE gAG Hochschule der Wirtschaft
Elmshorn, Germany
e-mail: sehring@nordakademie.de

***Abstract*—Model-driven software development is gaining attention due to the various benefits it promises. Typical approaches start with the modeling of application domains and continue with the specification of software to be developed. Model transformations are applied to develop and refine artifacts. In a final step, executable code is generated from models. Practice shows that model-based code generators have to bridge a rather large gap between the most refined software models and executable code implementing these models. This makes the development of code generators themselves an expensive task. In this article, we discuss ways to break down the development of code generators into smaller steps. Our discussion is guided on the one hand by principles of compiler construction and on the other hand by an application of model-driven development itself. Using a sample modeling language, we demonstrate how code generation can be organized to reduce development costs and increase reuse.**

Keywords*—*software development; software engineering; symbolic execution; top-down programming.

I. Introduction

Software construction requires methods and processes that guide development from an initial problem statement through all stages of the software lifecycle, culminating in the implementation, testing, roll-out, operations, and maintenance of the software.

Model-Driven Software Engineering (*MDSE*) strives to support such development processes by making explicit

- the artifacts created in each stage and possibly intermediate results
- the decisions that lead to the development of each artifact.

Ideally, MDSE supports the entire software lifecycle from requirements engineering and domain concepts through software architecture, design, and programming to software operations.

Figure 1 outlines some typical artifacts of software engineering processes. While many of them can be handled in MDSE processes, executable code must be generated for a particular target platform, such as a *Programming Language* (*PL*), software libraries, a runtime environment, and a target infrastructure. Later stages that depend on code, for example, operations tasks, also must be considered in code generation. This prepares code for activities like maintenance, monitoring, etc.

The support provided by MDSE approaches has advantages in many application areas. Models of sufficient formality can be checked for completeness or correctness to a certain extent. Traceability between artifacts allows to understand design decisions and model transformation steps during software maintenance. A final step of automated generation of executable code can save development costs during the implementation phase. Fully automated generation allows incremental development through model changes if the software is generated in an evolution-friendly manner.

Therefore, code generation from software models allows to take advantage of the potential benefits of modeling in MDSE. However, experience shows that generator development tends to be complex and costly. We see different reasons for this.

- The abstraction that models provide over programming language expressions require code generators to deal with a higher level of abstraction than compilers for PLs.
- Implementation details that are not reasonably part of software models must be added in code during generation.
- Various non-functional requirements of professional software development must be satisfied by generated code in addition to the requirements explicitly reflected by the software models. A code generator must add code for these as cross-cutting concerns.

Furthermore, these aspects of code generation typically require the development of project-specific generators.

Code generators are similar to compilers for high-level PLs. From this point of view, a model-driven process can be divided into a *frontend* and a *backend* part. In this logical division, the frontend deals with the more abstract models of the application domain and software design in *model-to-model transformations* (*M2MTs*). These early phases are covered by MDSE approaches. The backend activities of code generation, optimization, and target platform considerations are often hidden in implementations of comprehensive *model-to-text transformations* (*M2TTs*).

In this paper, we propose a structure for decomposing code generator development for easier development and a higher level of reuse.

The remainder of this paper is organized as follows: In Section II, we review model-driven software engineering with a focus on the final step of code generation. A corresponding approach to code generation is outlined in Section III. In Section IV, we illustrate the model-driven code generation approach with some sketches of code generation models. The paper is concluded in Section V.

II. Model-Driven Software Development

In this section, we revisit MDSE in general and code generation in particular in order to lay the foundation for the discussion of model-based code generation in the following sections.

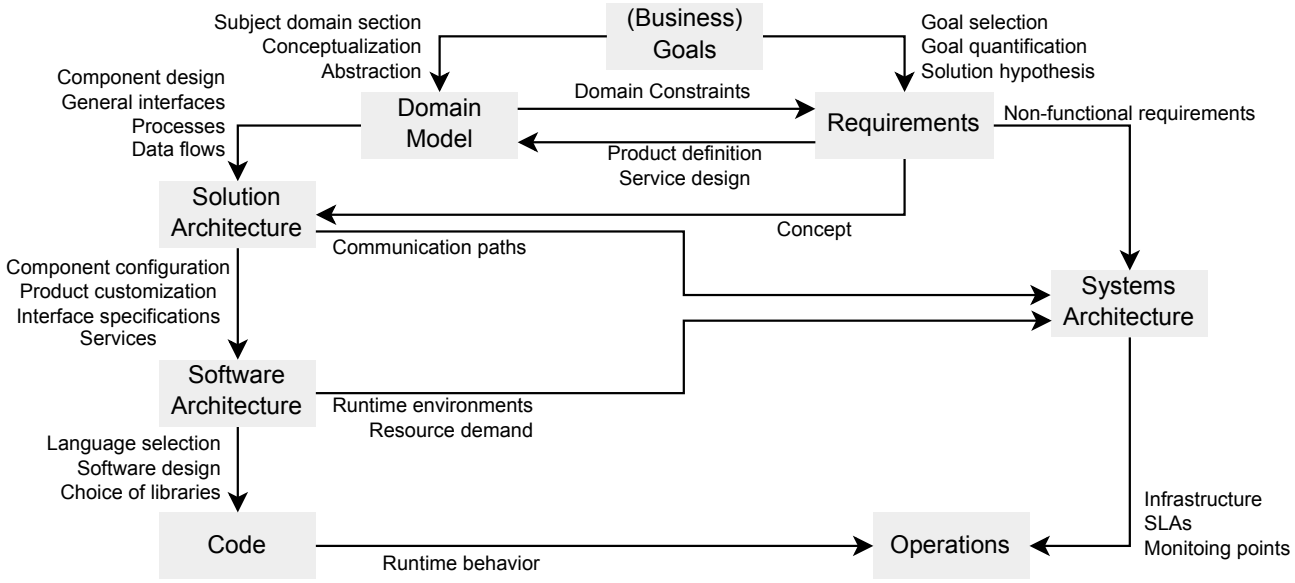

Figure 1. Typical software engineering artifacts.

A. Software Modeling

Models of the early steps in the software lifecycle are formulated from the perspective of the application domain. We do not consider domain models in this paper.

B. Cases of Code Generation

We see two application scenarios for software construction with MDSE:

1) approaches for systems of a given application class that share fixed functionality at some level of abstraction
2) approaches for application-specific functionality

A typical case of an application class with fixed functionality is the case of information systems, that typically provide only *Create, Read, Update, Delete* (*CRUD*) operations. Models of information systems, therefore, mainly represent domain entities and their relationships. Software generation is based on fixed patterns for code that provides CRUD functionality for the various entities.

Approaches that can be found in the class of generators that produce code with fixed functionality work with meta-programming [1], template-based approaches (see below), and combinations of these two [2]. Since generators for a specific target implementation can be built in a generic way, MDSE can be employed comparatively easy in this scenario.

In the general case of software containing custom business logic, software must be generated according to specified functionality. To automatically derive working software from specifications, MDSE approaches for application-specific business functionality must include formal models for precise definitions.

Means for deriving software from formal models are often built into editing tools for the respective formalisms. With respect to running software, formal models are typically used in one of two ways: Either code is generated from such models, or hand-written code is embedded in formal models at specific extension points.

For production-grade software systems, code generation is the only option in order to satisfy nonfunctional requirements.

A practical software system consists of different components, each of which is typically created by one generator each. Therefore, multiple code generators need to work in concert. To this end, different generator runs have to be orchestrated [2], and information exchange (for example, for identifiers used in different components) has to be managed [1].

C. Code Generation Techniques

Code is generated in a final step of an MDSE process, often based on M2TTs [3].

Special attention is paid to code generation, as this step can be well formalized in an MDSE process. There are several techniques for code generation, mainly generic code generators, meta-programming, and template-based techniques. Generative AI could be an alternative.

This way, there is reuse of software generators that translate formal specifications into code in a generic way. Typically, there is little or no way to direct the code generation for the case at hand [4]. Therefore, the generated code must be wrapped in order to be integrated into a production-grade software system, for example, to add error handling and additional code for monitoring.

a) Generic Code Generators: Custom functionality generally needs to be formulated in a Turing-complete formalism. Although the ability to verify such descriptions is limited, their expressiveness is required. Formal specifications of software functionality can be translated into working software by a code generator, that works like a compiler for a PL.

Code generators of modeling tools provide a well-tested and generally applicable translation facility. Specifications according to a given formalism are translated into a supported target environment. Examples include parser generators that generate code from grammars, software generators that take finite state machines as input [5], and those that use Petri Nets to execute code on firing transitions [6].

Generic code generators require significant development effort. But they can be developed centrally in a generic way. Therefore, there is a high degree of code reuse in the form of generators. However, the models used as input are application-specific, and they must be more elaborate than the input for other forms of generators.

b) Meta-Programming: Programs that generate programs are an obvious means of generating software. Meta-programming is possible with PLs, that allow the definition of data structures that represent code and from which code can be emitted. Since many widely used languages do not include meta-programming facilities, this capability is added through software libraries or at the level of development environments.

Meta-programming provides maximum freedom in generating custom code. Consequently, results can be tailored to the application at hand, including specific business logic.

However, the development of such generators tends to be costly, depending on the degree of individuality of code. This is due to the fact that meta-programs are harder to maintain and to debug due to their abstract nature. In addition, code reuse is very low for custom code.

c) Templates: Code with recurring structures can be formulated as templates with parameters for the variations of this uniform code. Code is generated by applying the templates with different parameter values.

A prominent example of a template-based approach is used for the *Model-Driven Architecture* (*MDA*). The *MOF Model to Text Transformation Language* [7] provides a means to define code templates based on (UML) models.

Templates are easy to write, depending on the degree of generics. They allow adaptation to the project at hand by making changes to templates. The degree of reuse of templates within a project can be high, depending of the structural similarities between parts of the code. Cross-project reuse can be expected to be quite low.

d) Generative AI: The currently emerging generative AI approaches based on large language models provide another way to generate code from descriptions. Based on a library of examples, they allow the interactive generation of code from less formal descriptions, especially natural language expressions.

Generative AI can deal with complex requirements and rules. It has the advantage of being able to generate code in multiple PLs from (almost) the same descriptions.

There are indications that generative AI may be particularly well suited to producing code on a small scale, for example, individual modules [8]. Final quality assurance and assembly currently remains a manual task.

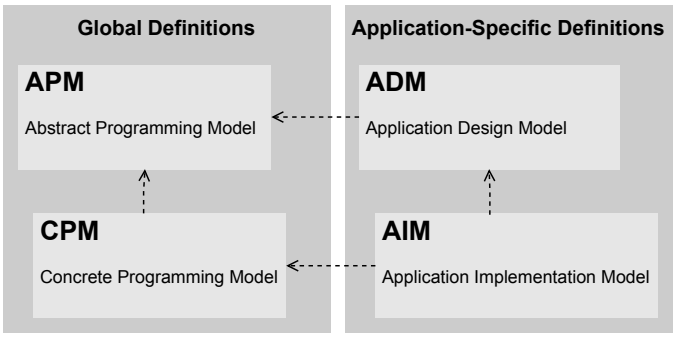

Figure 2. Code models and their relationships.

Instead of generating the actual software solution, generative AI can also be used to create code generators [3].

III. Model-Based Code Generation

In this paper, we discuss a way to construct code through a series of model refinement steps and final code generation. Thus, it follows the typical theme of M2MTs followed by an M2TT. However, our goal is to make the code generation step nearly trivial and fully automatic. To achieve this, we propose certain code models that bridge the gap between domain or solution models and executable code.

Our goal is to reduce the complexity of generators through abstraction and to reduce costs through reuse of *abstract code*.

Figure 2 gives an overview of the kinds of code models. Those in *Global Definitions* are provided centrally as a kind of modeling framework. Those in *Application-Specific Definitions* are models that are provided for each software project.

The four model boxes in the figure represent classes of models. There will be several concrete models for each of them.

We describe the models in the following subsections. Examples are given in the following main section.

The outline of the approach is as follows:

- Abstraction leads to a hierarchy of models.
- An *Abstract Program Model* (*APM*) provides a generic model of code.
- An *Application Design Model* (*ADM*) defines the functionality of a software system in terms of an APM.
- A *Concrete Program Model* (*CPM*) serves as a technology model; it maps an APM to a concrete implementation technology, such as a PL
- An *Application Implementation Model* (*AIM*) is used for code generation; it provides a project-specific association of the desired functionality and a technology model

With these models, some degree of reuse is achieved on the level of

1) programming models / building blocks of abstract programs
2) idioms and design patterns for refactoring and optimizing abstract programs
3) code generation from abstract representations of the constructs of a particular PL into code

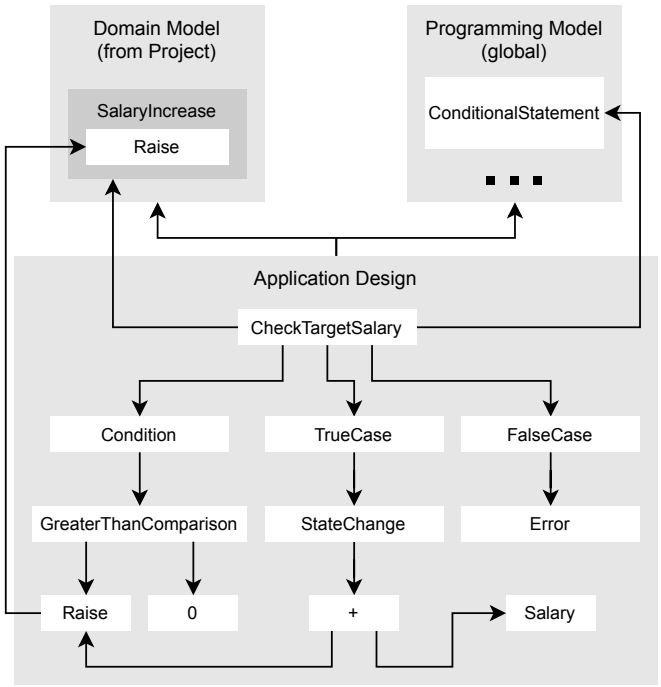

Figure 3. Typical software engineering artifacts.

A. Models of Programming

APMs serve as meta-models for abstract programs. Programming paradigms constitute a possible starting point for describing programming in general. Models of paradigms help to capture the essence of a class of PLs.

Properties of hybrid languages can be captured by combining models of programming paradigms. To this end, the modeling language used should allow models to be combined, and paradigm models must be set up to allow combinations.

There are differences between existing PLs that cannot be captured within one central model of programming. For example, object-oriented PLs have different ways of handling multiple inheritance. Therefore, there may be coexisting programming models, even for the same programming paradigm.

B. Assigning Functionality to Domain Models

In contrast to pure programming, program models in an MDSE process refer to more abstract models, especially those formulated from an application domain perspective. Program models result from M2MTs, or they refer to source models. Resulting model relationships are a basis for traceability [9].

Figure 3 illustrates a model relationship. A hypothetical domain model contains a *SalaryIncrease* concept with a *Raise* sub-concept . This specification is to be implemented using an imperative programming language, so there is, for example, a *ConditionalStatement*. The resulting model of the code for the software is represented as an ASM with a procedure *CheckTargetSalary*.

Application design models are essentially attributed syntax trees. In a kind of “reverse programming”, we manually construct syntax trees and generate code from them. This is not the right level for manual development of software generators. But program models can be derived from domain models similar to template-based software generation. The example in Figure 3 can be viewed this way. The advantage of abstract models and model relations over code templates is the independence from concrete programming languages. This allows us to make an early connection between a domain and a code model while still having the option of choosing the implementation details, including the programming language or other implementation technologies to be used.

C. Stepwise Refinement of Programming Models

Concrete models define which language constructs are available in a particular PL that is selected for implementation. For code generation, the abstract application code (ASM) is combined with a CPM containing models of typical programming language constructs / idioms etc. in generalized form. M2MTs are applied to the combined model to transform it into an AIM that is suitable for code generation.

Model transformation consists of refining abstract program models with respect to a concrete PL or other implementation technology. There are several reasons why concrete models differ from abstract programming models. For example, there are different ways to implement abstract code in concrete PLs, similar PLs may have different best practices, they may have different constraints, and they may require different optimizations.

The transformation from an abstract to a concrete program need not be done in one step. For example, there is typically a hierarchy of abstractions, from abstract programs at the programming paradigm level, to classes of PLs and PL families, to concrete PLs, PL implementations or dialects, or even project-specific style guides.

D. Generating Code from Abstract Programs

An AIM is a model of a program that is suitable for code generation. This means, that all parts of a model are assigned a concrete PL expression and thus a syntactic form.

With this model property, code can be generated by assembling pieces of code that each represent model entities.

IV. Examples of Model Definitions for Code

We outline some models in order to illustrate the approach presented in the previous section. We use the *Minimalistic Meta Modeling Language* (*M³L*) as our modeling notation. This language is briefly introduced in order to discuss some model sketches.

A. M³L Overview

The M³L is a (meta) modeling language that has been reported about. Definitions are of the general form
A `is a` **B** `{` **C** `is a` **D** `}` `|=` **E** `{` **F** `}` `|-` **G H** `.`
Such a statement matches or creates a *concept A*. All parts of such a statement except the concept name are optional.

The concept *A* is a *refinement* of the concept *B*. Using the “is the” clause instead defines a concept as the only specialization of its base concept.

```
Type
Boolean is a Type
True is a Boolean False is a Boolean
Integer is a Type
Variable { Name Type }
Procedure { FormalParameter Statement }
Statement
ConditionalStatement is a Statement {
  Condition     is a Boolean
  ThenStatement is a Statement
  ElseStatement is a Statement }
Loop is a Statement { Body is a Statement}
HeadControlledLoop is a Loop {
  Condition is a Boolean }
VariableDeclaration is a Statement {
  Variable InitialValue is an Expression }
ProcedureCall is a Statement {
  Procedure ActualParameter}
Expression is a Statement ...
```

Figure 4. Sample base model of procedural programming.

```
IfTrueStmt is a ConditionalStatement {
  True is the Condition
} |= ThenStatement
IfFalseStmt is a ConditionalStatement {
  False is the Condition
} |= ElseStatement
```

Figure 5. Semantics of conditional statements.

The concept *C* is defined in the *context* of *A*; *C* is part of the *content* of *A*. Each context defines a scope, and scopes are hierarchical. Concepts like *A* are defined in an unnamed top-level context.

There can be multiple statements about a concept with a given name in a scope. All visible statements about a concept are cumulated. This allows concepts to be defined differently in different contexts.

Semantic rules can be defined on concepts, denoted by "|=". A semantic rule references another concept, that is returned when a concept with a semantic rule is referenced.

Context, specializations, and semantic rules are employed for *concept evaluation*. A concept evaluates to the result of its syntactic rule, if defined, or to its *narrowing*. A concept *B* is a narrowing of a concept *A* if

- *A* evaluates to *B* through specializations or semantic rules, and
- the whole content of *A* narrows down to content of *B*.

Concepts can be marshalled/unmarshalled as text by *syntactic rules*, denoted by "|-". A syntactic rule names a sequence of concepts whose representations are concatenated. A concept without a syntactic rule is represented by its name. Syntactic rules are used to represent a concept as a string as well as to create a concept from a string.

```
Expression
Value is an Expression
ConditionalExpression is an Expression {
  Condition  is an Expression
  TrueValue  is an Expression
  FalseValue is an Expression }
Function is a Value {
  FormalParameter FunTerm }
FunCall is an Expression {
  Function ActualParameterList }
PartialFunCall is a FunCall, a Value
```

Figure 6. Sample base model of functional programming.

```
MetaClass is an Object { Method }
Method { Parameter is an Object }
Classifier is a MetaClass
Interface is a Classifier
AbstractClass is a Classifier
ConcreteClass is a Classifier
ObjectClass is a ConcreteClass
Object is an ObjectClass
```

Figure 7. Sample base model of object-oriented programming.

B. Example Programming Models

Sticking with the example of starting the modeling of programming with programming paradigms, there may be models that describe typical constructs of PLs of a particular paradigm. We give short outlines of PL base models for the most important programming paradigms. Many details, such as any type system, are omitted.

1) Procedural Programming: Descriptions of some typical constructs of imperative PLs are shown in Figure 4. Typical control flow constructs, such as conditional statements and loops are given as M³L concepts.

For model checking or for model execution, the language constructs must be given semantics. For example, the behavior of the *ConditionalStatement* can be defined as shown in Figure 5. When evaluated, such a conditional statement will match (become a derived subconcept) of either *IfTrueStmt* or *IfFalseStmt*, depending on which concept a given *Condition* evaluates to. The semantic rule is inherited from the derived base concept, making the statement evaluate to either the "then branch" or the "else branch".

This way of attaching semantics is typical for M³L models; other modeling languages may have different ways of attaching semantics. We will not go into this in detail. However, it is an important part of the PL base models.

2) Functional Programming: Figure 6 outlines base definitions for functional PLs. Note that this model contains definitions that may not apply to all functional PLs, so other APMs exist.

3) Object-Oriented Programming: Only some base definitions for class hierarchies at the instance and class levels are sketched in Figure 7. The complete model is much more

```
Vi is a Variable {
 i is the Name Integer is the Type }
VariDeclaration is a VariableDeclaration {
 Vi is the Variable 0 is the InitialValue}
SomeLoop is a WhileLoop {
 LessThanIntComparison is the Condition {
  Vi is the Left 10 is the Right }
 VariableAssignment is the Body {
  Vi is the Variable
  IntegerSum is the Expression {
   Vi is a Summand 1 is a Summand }}}
```

Figure 8. A sample abstract program.

elaborate, and there are even more variants of PLs than in the other paradigms.

C. Abstract Programs

ADMs can be formulated in the M³L as refinements of APMs. Figure 8 shows an example of imperative code for a loop that increments a variable i from 0 to 9.

D. Abstract Program Transformations

In our experimental setup with the M³L, model transformations can be expressed by relating concepts to each other: one is a refinement or a redefinition of the other. In other modeling languages, the respective model transformation or model evolution facilities are used.

E. Code Generation

The final M2TTs to produce source code are performed on models that combine an ADM with the abstract program for the problem at hand and a CPM that declares concrete PL constructs.

The CPM comes with predefined translation tables that are used to generate code. Such translation tables can be formulated by syntactic rules in the example of the M³L.

For example, rules for language-dependent code generation for two different languages can be such as:

```
Java is a ProgrammingLanguage {
 ConditionalStatement
  |- if ( Condition )
     ThenStatement
     ElseStatement . }
Python is a ProgrammingLanguage {
 ConditionalStatement
  |- if Condition :
     " " ThenStatement
     else:
     " " ElseStatement . }
```

By separating APMs and CPMs, it is possible to generate different code from the same abstract program. In the M³L, concepts can easily be redefined with different syntactic rules in the context of a PL. When generating code in such a PL context, the rules of all language constructs for that PL are used. Variations for language dialects can be handled in subcontexts where some rules are redefined.

V. Conclusion and Future Work

Model-Driven Software Engineering is receiving a lot of attention for the benefits it brings to software engineering processes. While model-to-model and model-to-text transformations are being researched, in practice the final step of code generation from models is too costly to be applied in many application scenarios.

In this paper, we propose an approach to define code generators for the MDSE toolchain. Code generators consist of executable models that are defined step by step, where each step is characterized by a simple model. In addition, some models are generic and can be shared. If the models that define a code generator are formulated in the same modeling framework as the models for earlier stages of the software engineering process, then models of the application domain and models of the software can be closely related.

The proposed approach allows us to achieve the goals of reduced development costs for code generators and of increased reuse. Using multiple levels of abstraction makes each development step easier and less expensive. Since the most abstract models are generically applicable, they can be reused across applications.

Future work includes experiments with real-world code models before pursuing new research directions. Since many important PLs are hybrid in nature, remaining issues with combined APMs need to be addressed, such as the mismatch between imperative and declarative PLs.

Acknowledgements

The publication of this work was made possible by the NORDAKADEMIE gAG.

References

[1] H.-W. Sehring, S. Bossung, and J. W. Schmidt, “Content Is Capricious: A Case for Dynamic System Generation,” Proceedings Advances in Databases and Information Systems, Springer, 2006, pp. 430–445.

[2] H. Mannaert, K. De Cock, and J. Faes, “Exploring the Creation and Added Value of Manufacturing Control Systems for Software Factories,” Proceedings Eighteenth International Conference on Software Engineering Advances, ThinkMind, 2023, pp. 14–19.

[3] K. Lano and Q. Xue, “Code generation by example using symbolic machine learning,” SN Computer Science, vol. 4, Springer Nature, 2023.

[4] T. Mucci. *What is a code generator?*, Think 2024, [Online] Available from: https://www.ibm.com/think/topics/code-generator. 2024.6.28.

[5] T. E. Shulga, E. A. Ivanov, M. D. Slastihina, and N. S. Vagarina, “Developing a software system for automata-based code generation,” Programming and Computer Software, vol. 42, pp. 167—173, 2016.

[6] K. Radek and J. Vladimír, “Incorporating Petri Nets into DEVS Formalism for Precise System Modeling,” Proceeding Fourteenth International Conference on Software Engineering Advances, ThinkMind, 2019, pp. 184–189.

[7] Object Management Group. *MOF Model to Text Transformation Language, v1.0*, OMG Document Number formal/2008-01-16, [Online] Available from: https://www.omg.org/spec/MOFM2T/1.0/PDF. 2024.7.4.

[8] M. Harter, “LLM Assisted No-code HMI Development for Safety-Critical Systems,” Proceedings Sixteenth International Conference on Advances in Human-oriented and Personalized Mechanisms, Technologies, and Services, ThinkMind, 2023, pp. 8–18.

[9] S. Hajiaghapour and N. Schlueter, “Evaluation of different Systems Engineering Approaches as Solutions to Cross-Lifecycle Traceability Problems in Product Development: A Survey,” Proceedings International Conference of Modern Systems Engineering Solutions, ThinkMind, 2023, pp. 7–16.