Model-Based Development of Code Generators for Use in Model-Driven Development Processes

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 as a software engineering approach due to the various benefits it offers. Typical approaches start with the modeling of the application domain at hand and continue with the specification of the software to be developed. Results are documented by specific software engineering artifacts. Especially in model-driven approaches, these artifacts are formal or semi-formal models. Model transformations are applied to develop and refine artifacts. In a final step, code is generated from the models. Code can be source code written in specific programming languages, configuration files, and the like. Practice shows that model-based code generators have to bridge a rather large gap between the most refined software models and the compilable code that implements these models. This makes the development of code generators itself an expensive task. In this article, we discuss ways to break down the development of code generators into smaller steps. Our discussion is guided by both principles of compiler construction and by an application of model-driven development itself. Using a modeling language, we demonstrate how code generation can be organized to reduce development costs and increase reuse. In addition, program code becomes part of the model transformation sequence, allowing code changes to be automated and model elements to be referenced from code.**

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, rollout, operation, and maintenance of the software.

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

- the artifacts created at 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 the code, such as operations tasks, must also be considered in code generation. This prepares the code for activities such as 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 understanding of 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 several reasons for this.

- The abstraction that models provide over programming language expressions requires 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 article, we propose a structure for decomposing code generator development for easier development and a higher level of reuse.

This article extends the presentation of a conference paper on the topic [1].

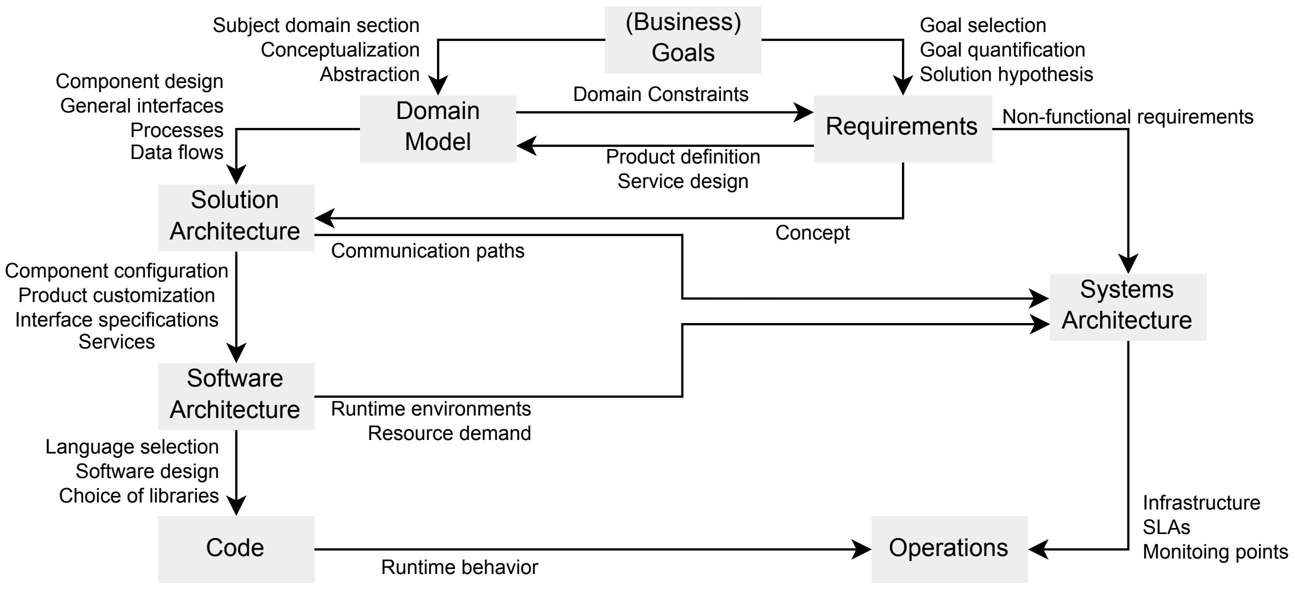


Figure 1. Typical software engineering artifacts.

The remainder of this article 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. We use a meta modeling language to present some models as experiments with the approach. This language is introduced in Section IV. In Section V, we illustrate the model-driven code generation approach with some sketches of code generation models. Some remarks on the derivation of code models from more abstract models are made in Section VI. The paper is concluded in Section VII.

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.

A. Subject Domain Modeling

With few exceptions, the purpose of software is to solve some a real-world problem. For example, software is used to perform scientific calculations, to support a company's business processes, or to control hardware. As a result, a software project is typically embedded in a project with a broader scope. Therefore, the analysis and documentation of the task at hand begins with entities that lie outside the software domain, but within the broader *subject domain*, *application domain*, or simply *domain*.

The purpose of a domain model is twofold: it clarifies the terms and rules of the (real-world) application scenario and it defines a possible solution to the task at hand in terms of the application domain. As a model, it furthermore provides an abstraction by defining the section of the application domain that is considered by the software to be built.

Figure 1 presents typical stages of a project by naming the artifacts under consideration. Typically, a project begins by setting goals. Goals define success criteria for the overall project. Goals guide the choice of abstraction for the domain model, which is also defined in an early stage of a project. The same goes for the requirements (for the software), which are derived from both the goals and the domain conceptualization.

Subject domain modeling is beyond the scope of this article.

B. Software Modeling

Models of the early steps in the software lifecycle are formulated from the perspective of the application domain. From these, models of software from a technical perspective are derived. The solution architecture typically is the link between the domain perspective and the software perspective (see Figure 1).

Software description spans a series of models, starting from abstract ones to increasingly concrete ones. M2MTs are applied to derive models, even though mainly design decisions drive the modeling process.

When software models reach a sufficiently concrete level, code is finally emitted by M2TTs. The code generation and maintenance part is often not well represented in an MDSE process, though. On top of that, in most cases textual representations of code are decoupled from the models of earlier design phases.

Ideally, models of the software also support the operations and maintenance phase. In this case, they must be available at runtime [2].

The more abstract descriptions of software at the level of software architecture specifications are not in the focus of this article. However, it depends on the type of M2TTs or code generators at which point the transition from abstract software models to concrete source code takes place.

C. 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 [3][4], template-based approaches (see below), and combinations of these two [5]. 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. Depending on the modeling approach chosen, a model interpreter may not provide sufficient performance or scalability at runtime. The coexistence of the higher-level model and the lower-level PL code can increase maintenance complexity due to the different roles involved. Since changes to a model can affect the code [6], it is crucial to be able to trace of code design decisions.

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 [5], and information exchange (for example, for identifiers used in different components) has to be managed [4].

D. Code Generation Techniques

Code is generated in a final step of an MDSE process, often based on M2TTs [7].

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 [8]. 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 [9], and those that use Petri Nets to execute code on firing transitions [10].

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 [11]. 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 [12]. 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* [13] 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.

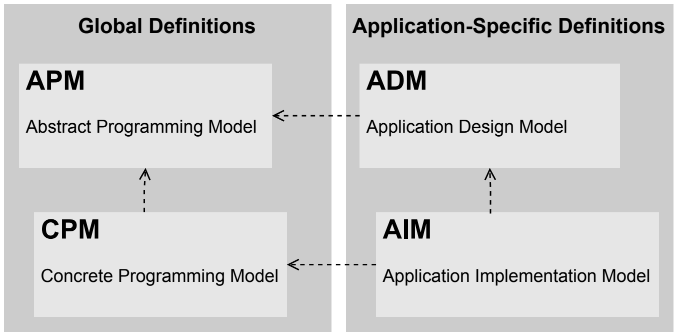

Global Definitions
Application-Specific Definitions
APM
Abstract Programming Model
ADM
Application Design Model
CPM
Concrete Programming Model
AIM
Application Implementation Model

Figure 2. Code models and their relationships.

d) Generative AI: The 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 [14]. Final quality assurance and assembly currently remains a manual task.

Instead of generating the actual software solution, generative AI can also be used to create code generators [7].

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

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 [15].

Figure 3 illustrates a model relationship. A hypothetical domain model contains a *SalaryIncrease* concept with a *Raise* subconcept . 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 ADM 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.

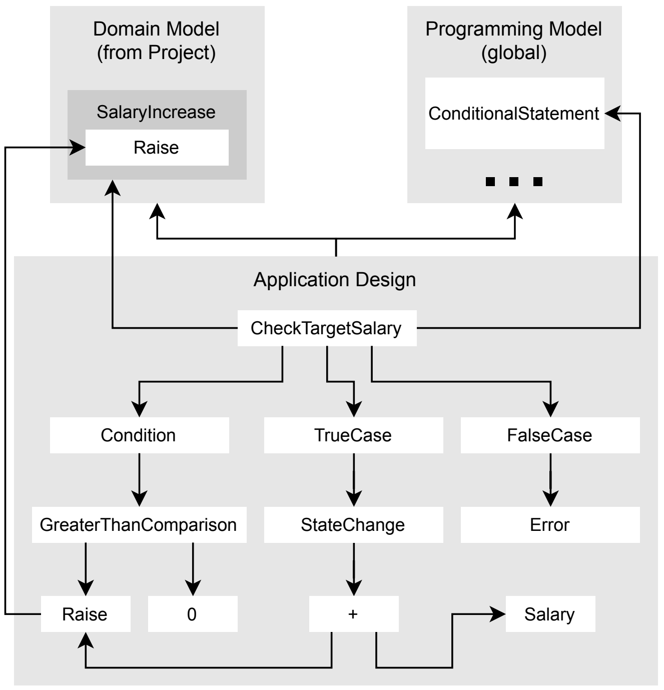


Figure 3. Typical software engineering artifacts.

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 (given as an ADM) 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. M^{3}L Overview

We use the *Minimalistic Meta Modeling Language* (*M^{3}L*) as our modeling notation. This language is briefly introduced in this section in order to discuss some model sketches in the remainder of the article.

A. Basic Concept Definitions

The M^{3}L is a (meta) modeling language that has been reported about. It is minimal in the sense that it is designed with a sparse syntax that is oriented towards metalogic and a simple semantics that basically breaks down to set theory.

A model in the M^{3}L consists of a series of definitions of *concepts*. A concept definition, in its simplest form, just names one concept:

```
A
```

Naming a concept leads to its evaluation (see Section IV-D below), if it exists, or to its creation otherwise. In the simplest form, just the name is introduced. Therefore, in either case, the concept A is defined after the above statement.

A concept can be defined as a *refinement* of another. For example, the following statement defines the concept A as a refinement of the concept B using the “is a” / “is an” clause.

```
A is a B
```

A is also called a *subconcept* of B, B the *base concept* or *generalization* of A. Multiple base concepts can be given at once:

```
A is a B, an E
```

Refinements *inherit* the definition from their base concept. This includes base concepts, content (see below), and rules (see below).

Using the “is the” clause instead defines a concept as the only specialization of its base concept.

```
C is the D
```

The concept C may have further base concepts, but D has no refinements other than C.

There may be multiple definitions of one concept. If definitions refer to the same concept name, their effect is cumulated.

```
A is a  B
...
A is an E
```

defines A as a subconcept of both B and E.

B. Contextual Concept Definitions

A concept C is defined in the *context* of a concept A by a definition of the form

```
A is a B {
  C is a D
}
```

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. A concept A is *visible* in the context of a concept C if A belongs to the content of C or if there is a concept B such that A is visible in the context of B and C belongs to the content of B.

There can be multiple statements about a concept with a given name in different scopes. Contextual definitions are called *redefinitions*. All visible statements about a concept are cumulated. This allows concepts to be defined differently in different contexts. For example, the statements

```
A { C is a D }
C
```

define C as a specialization of D in the context of A, but without base concept in the topmost context.

A concept in a nested context is referenced as

```
C from A
```

There are well-defined names that refer to a part of a concept: "the concept" refers to the concept of a current definition, "the name" to its name, and "the content" refers to all content concepts of a concept. These are particularly used in refinements and redefinitions, such as

```
ListOfThings {
  Head is a Thing
  Tail is a ListOfThings
  AddToList {
    Elem is a Thing
  } is a ListOfThings {
    Elem is the Head
    the concept is the Tail
  }
}
```

This code allows adding a element to a singly linked list by using a current list as the tail of a new list (last line).

C. Semantic Rules

Semantic rules can be defined on concepts, denoted by a double turnstile ($\models$), in code written as "|=". A semantic rule references another concept, that is returned when a concept with a semantic rule is referenced. Like for any other reference, a non-existing concept is created on demand.

The scope of a semantic rule is specific: concepts referenced by the rule are resolved in the context of the concept to which the semantic rule belongs. If the rule leads to the creation of a new concept. then this concept is placed in the same context as the concept carrying the rule.

An example of a semantic rule is as follows:

```
A is a B {
  C is a D
} |= E { C }
```

In this example, E is defined by the semantic rule. Its content C is taken from the content of A. Because of the scoping rules, C must also be declared in the context of A. Otherwise, the rule is considered erroneous.

D. Concept Evaluation

Context, refinements, and semantic rules are employed for *concept evaluation*. When an existing concept is referenced, it is first evaluated, and the result of the evaluation is returned.

A concept evaluates to the result of its semantic rule, if defined, or else to its *narrowing* as defined below.

A concept's semantic rule is determined by the following steps:

1) If the concept's narrowing (see below) has a semantic rule, then this one is taken.
2) Else, if the narrowing inherits a semantic rule, then the rule of the closest ancestor is used.
3) Else, the set of *derived base concepts* (see below) is checked for semantic rules.
4) Else, there is no semantic rule.

A concept B is a narrowing of a concept A, if B is a transitive "is the" refinement of A.

A concept B is a derived subconcept of a concept A if

- the set of (transitive) base concepts of B is a superset of the set of (transitive) base concepts of A, and
- the content of A narrows down to content of B; this means that for every concept C in the content of A, there exists a concept D in the content of B such that D is C or one of its narrowings.

To evaluate a concept, syntactic rules and narrowing are applied repeatedly, until a concept evaluates to itself. Infinite evaluation loops are considered erroneous definitions.

An example of concept evaluation is provided by Figure 8 in Section V-B.

E. Syntactic Rules

Concepts can be marshaled/unmarshaled as text by *syntactic rules*, denoted by a turnstile ($\vdash$), in code represented by "|-". A syntactic rule names a sequence of concepts whose representations are concatenated, ended by a dot.

The representation of each concept is in turn defined by the syntactic rule of its evaluation. 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; they define both an output and a parser.

As an example, consider a definition

```
A is a B {
  C is a D
} |- the name says C .
```

When producing output for

```
John { hello is the C } is an A
```

it produces "John says hello", since *John* inherits the syntactic rule. Note that previously undefined concepts like *hello* and *says* are defined on spot, and that they are represented by their name.

```
TypedPL is a ProgrammingLanguage {
  Type

  Boolean is a Type
  True is a Boolean
  False is a Boolean

  Integer is a Type {
    Succ is a PositiveInteger {
      the concept is the Pred }
  }
  0 is an Integer
  PositiveInteger is an Integer {
    Pred is an Integer
  }
  1 is a PositiveInteger { 0 is the Pred }
}
```

Figure 4. Sample base model of typed programming languages.

V. Examples of Model Definitions for Code

We outline some models in order to illustrate the approach presented in Section III. We use the M^{3}L as introduced in the previous section as our modeling notation.

A. *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 that may provide concepts for APMs. Many details are omitted for the discussion in this article.

As the basis of programming models, assume base concept *ProgrammingLanguage*. Different aspects of programming are derived from that concept.

As a first specialization, Figure 4 sketches some basic concepts for typed programming languages. The concept *Boolean* represents a type for Boolean values; this type as the finite set of values *True* and *False*. The type *Integer* provides a definition of numbers based on the Peano axioms. Even though concrete programming languages offer a builtin type for integers that supports low-level operations provided by hardware, we want APMs to have provable properties independently of any concrete implementation. Please note that *Succ* is a method of an *Integer* object that answers the successor; it will be created if it does not exist. In contrast, *Pred* is an attribute that is set explicitly, for example, by *Succ*. Other types are omitted in this article, but may be defined accordingly.

1) Procedural Programming: Descriptions of some typical constructs of imperative PLs are shown in Figure 5. Typical control flow constructs, such as conditional statements and loops are given as M^{3}L concepts.

2) Functional Programming: Figure 6 outlines the basic definitions for functional PLs. Note that this model contains

```
ImperativeProgramming is a TypedPL {
  Variable {
    Name
    Type from TypedPL }

  Statement
  Sequence is a Statement {
    Statements is a Statement }
  CompoundStatement is a Sequence
  ParallelExecution is a Statement {
    ConcurrentStatements is a 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 from TypedPL }
  CountingLoop is a Loop {
    Counter    is a  Variable {
      Integer is the Type }
    LowerBound is an Integer from TypedPL
    UpperBound is an Integer from TypedPL
    Step       is an Integer from TypedPL}
  VariableDeclaration is a Statement {
    Variable
    InitialValue is an Expression }

  Expression is a Statement
  Value is an Expression
  VariableReference is an Expression
  UnaryExpression is an Expression {
    Operand is an Expression }
  BinaryExpression is an Expression {
    Operand1 is an Expression
    Operand2 is an Expression }

  ...
}

ProceduralProgramming
  is an ImperativeProgramming
{
  Procedure {
    FormalParameter is a Variable
    Body            is a Statement }

  ProcedureCall is a Statement {
    Procedure
    ParameterBinding {
      FormalParameter from Procedure
      ActualParameter is an Expression } }

  ...
}
```

Figure 5. Sample base model of procedural programming.

```
FunctionalProgramming {

  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
    ParameterBinding {
      FormalParameter from Function
      ActualParameter is an Expression
    }
  }

}
```

Figure 6. Sample base model of functional programming.

definitions that may not apply to all functional PLs, so other APMs exist.

The simple syntax of functional PLs makes the definition of this programming paradigm quite short. Note, however, that many properties of functional PLs are covered by the model. For example, partial function application is expressible since a function call (*FunCall*) is an *Expression*, and a *Function* is a *Value* which is also an *Expression*. So function calls can deliver functions. Likewise, higher-order functions are covered by the model.

The model in Figure 6 omits typical libraries of predefined functions, for example, the various kinds of recursive higher-order functions. These differ slightly between concrete PLs, though, so there will be variations.

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

In particular, typical object-oriented PLs consist of two "sub-languages". In a declarative part, objects or classes are defined, method signatures (message formats) are declared, etc. Executable code is mainly found in method bodies, which are implemented depending on the kind of object-oriented PL used.

Statements as used in imperative programming are one way of implementing methods. In Figure 7, this is sketeched by the *OOP-imperative* concept. In particular, compound statements are typically used. In contemporary PLs, specific additions to the set of statements cater for object-oriented structures.

```
ObjectOrientedProgramming {

  MetaClass is an Object { Method }
  Method {
    Parameter is an Object
    Body
  }

  Classifier is a MetaClass
  Interface     is a Classifier
  AbstractClass is a Classifier
  ConcreteClass is a Classifier

  ObjectClass is a ConcreteClass
  Object is an ObjectClass
}

OOP-imperative
  is an ObjectOrientedProgramming,
     an ImperativeProgramming
{
  Method {
    LocalVariables is the Parameter,
                        a   Variable
    Body is a Statement
  }
}

OOP-functional
  is an ObjectOrientedProgramming,
     a  FunctionalProgramming
{
  Method {
    Body is a Function {
      FormalParameter is the Parameter
    }
  }
}
```

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

A second option of implementing methods is in a functional way by assigning a function to a message. This is outlined by the concept OOP-functional in Figure 7. The parameter(s) of the function (*FormalParameter from Function*) that is used as a method body are passed from the parameters of the method (*Parameter from Method*).

B. Programming Language Semantics

For model checking or for model execution, language constructs as outlined in the previous subsection must be given semantics. The semantics of specific PLs is abstracted so that different generalized APMs can be defined to capture the various interpretations of PL constructs.

As an example, the behavior of the *ConditionalStatement* can be defined as shown in Figure 8. A concrete conditional statement as part of a program may look like

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

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

Figure 8. Semantics of conditional statements.

```
MyConditional is a ConditionalStatement {
  SomePredicate is the Condition
  Statement1    is the ThenStatement
  Statement2    is the ElseStatement
}
```

A concrete conditional statement like *MyConditional* must not have a semantic rule defined.

When evaluated, such a conditional statement will match (become a derived subconcept) of either *IfTrueStmt* or *IfFalseStmt*, depending on what *SomePredicate* evaluates to: *MyConditional*, like any refinement of *ConditionalStatement*, has a superset of the base concepts of both *IfTrueStmt* and *IfFalseStmt* since it is an explicit subconcept. The content of which one of them matches is determined by the evaluation of *SomePredicate* that should yield either *True* or *False*. Then, exactly one of *IfTrueStmt* or *IfFalseStmt* will be a derived base concept of *MyConditional* which will inherit the semantic rule that leads to the correct interpretation of the conditional statement.

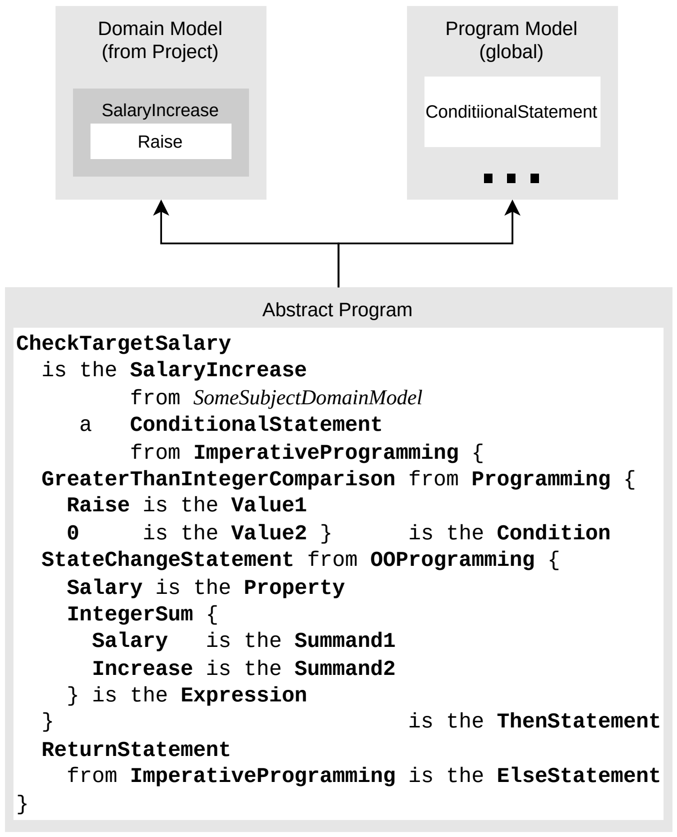
The semantic rule is inherited from the derived base concept, making the statement evaluate to either the "then branch" or the "else branch".

This way of attaching semantics is typical for M^{3}L 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.

C. Abstract Programs

Based on the programming concepts outlined in the preceding subsections, abstract programs can be formulated using "instances" of these concepts. This means that refinements of the concepts of an APM form an ADM. Such a program is abstract in the sense that it is not written in a concrete PL. It is more like an attributed abstract syntax tree.

Figure 9 shows an example of an abstract code fragment for a sample ADM. The code is based on the model shown in Figure 3. An imperative object-oriented programming style is chosen. The entire code fragment represents a conditional statement that checks for a positive raise. The *ConditionalStatement* is outlined in Figures 10 and 8. The *GreaterThanIntegerComparison* may be a binary predicate that compares integers. It is used as the condition. The two branches of the conditional statement, *ThenStatement* and *ElseStatement*, are set to accordingly. On a positive raise, there is a state change, assuming that there is some employee object (omitted from the example; *Raise* and *Salary* may be declared outside the code fragment). On an invalid raise, the code is simply exited.



```
CheckTargetSalary
  is the SalaryIncrease
         from SomeSubjectDomainModel
     a   ConditionalStatement
         from ImperativeProgramming {
  GreaterThanIntegerComparison from Programming {
    Raise is the Value1
    0     is the Value2 }      is the Condition
  StateChangeStatement from OOProgramming {
    Salary is the Property
    IntegerSum {
      Salary   is the Summand1
      Increase is the Summand2
    } is the Expression
  }                            is the ThenStatement
  ReturnStatement
    from ImperativeProgramming is the ElseStatement
}
```

Figure 9. Typical software engineering artifacts.

This allows complete code bases to be formulated in an abstract way. Starting from a domain model, programming constructs can be introduced step by step to from an ADM. Using a CPM, an ADM can be transformed into an AIM.

D. Abstract Program Transformations

In our experimental setup with the M^{3}L, model transformations can be expressed by relating concepts to each other. In other modeling languages, the respective model transformation or model evolution facilities are used [16][17][18]. In the M^{3}L, M2MTs can be implemented by concept refinement, concept redefinitions, and semantic rules. M2TTs are expressed by syntactic rules in combination with concept evaluation.

Model transformations are, in the discussion of this article, transformations of abstract programs. The advantages of expressing code in abstract forms are manifold.

1) Stepwise Concretization: Many aspects of an implementation must be considered at once when there is just one code generation step. Code generation does not only have to produce code with the correct functionality. It must also respect nonfunctional requirements. On top of that, there are cross-cutting concerns like error handling when producing executable code.

2) Higher Degree of Reuse: Abstract code has a possibly higher chance of being reusable. In particular, concepts may serve as prototypes that are redefined to concrete uses. On

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

Figure 10. A sample abstract program.

top of that, high-level designs, such as design patterns can be codified an applied where needed [19].

3) Roundtrip Engineering: When maintaining software, traceability from models to code is improved over M2TTs that generate code directly from more abstract models. If a need for change can be localized in the working software, it is potentially easier to trace it back to models.

4) Optimization: Code optimization is more effective at higher levels of abstraction. For example, algorithmic changes will usually have a greater impact than local code optimizations. With the ability to optimize code at each model layer, any generated code will benefit from optimizations without having to rely on PL-specific tools, such as compilers.

5) Cross-Platform Development: Finally, abstract programs allow target code to be generated in different PLs. This facilitates the development of distributed applications in heterogeneous environments, and the generation of code for different platforms from the same (abstract) code base.

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 M^{3}L.

For example, rules for language-dependent code generation for two different languages can be such as sketeched in Figure 10.

By separating APMs and CPMs, it is possible to generate different code from the same abstract program. In the M^{3}L, 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.

```
Company {
  Person { Name Age }
}
CompanyImpl is a Company {
  PersonRecord is a Person,
                  a Record from TypedPL {
    Name is a  String  from TypedPL
    Age  is an Integer from TypedPL
  }
}
```

Figure 11. Sample of a first application design model for a domain model.

VI. Higher Programming Abstractions

In this article, we focus on models of programming language code. In an overarching modeling process, there are M2MTs that lead from domain models to software models and ones that lead from general software models to code models. Such M2MTs mark the start of a new development phase.

Depending on the models chosen, the gap between models of two subsequent phases may be rather large. Reasons are the change of concepts and the preferred structures that combine them. In this section, we briefly discuss two kinds of models that may bridge the gap more gently in the following two subsections.

Intermediate models that contain concepts from both the application domain as well as the software domain allow introducing a subset of the required technical concepts.

There are certain abstractions of code that provide a starting point for such intermediate models [3]. In particular, software design approaches for manual software development can be used as blueprints for the creation of initial code models.

A. Domain Models as Initial Application Design Models

An advantage of a consistent (meta) modeling approach like the one provided by the M^{3}L is the seamless transformation of domain-centric models towards code-centric models. This may be achieved by hybrid models, such as high-level code models that are based on domain models as data or object models and that constitute a first ADM. ADMs can be formulated in the M^{3}L as refinements of APMs. Implementation aspects are added stepwise to transform such a hybrid model into an adequate ADM.

Hybrid models provide more abstract software constructs in the sense of *Domain-Driven Development* or *Domain-Specific Languages* (*DSLs*) in which implementations relate to domain concepts directly [20]. They also provide a domain model that includes a connecting points to an implementation that may alternatively be added in a generic way. For information system, for example, typically CRUD operations are added for a domain-specific data model (see Section II-C). In such an approach, a first model provides consistent technical types for the attributes of (domain) entities and operations defined on them.

```
Company {
  OrgUnit is a Record {...}
  BusinessUnit is an OrgUnit {
    Departments is an Department
  }
  Department   is an OrgUnit {
    Teams is a Team
  }
  Team         is an OrgUnit {
    Members is a Person
  }
}

CompanyImpl is a Company

OrgUnitImplementation
  is a CompositePatternApplication
{
  CompanyImpl is the TargetModel
  OrgEntity   is the ComponentClass
  OrgUnit     is the CompositeClass
  Person      is the LeafClass
  Members is the AggregationRelationship
}
```

Figure 12. A sample domain model and design pattern application.

For an example, please consider a concept *Person* that is part of a domain model *Company*. Data about *Person*s shall be managed by an aggregated type that consist of a person's *Name* and *Age*. A first ADM for the example is outlined in Figure 11 as *CompanyImpl*, where persons are modeled as *Record*s composed of a *String* and an *Integer*. This type information is assigned on the basis of abstract type information as sketched in Figure 4.

Depending on the choice for a technology, there will be a direct mapping of the data types, and (CRUD) functionality can be added based in that mapping (see Section II-C).

B. Design Patterns

Design patterns provide proven solutions for specific tasks [21]. Additionally, they provide design standards that are well known amongst developers. This additional use of patterns may be codified in model transformations. There are patterns that are helpful in elaborating code models, and ones that help bridging the gap from more abstract to more concrete models.

As an example, assume organizational units in a company defined by a domain concepts in Figure 12. This sample company has three organizational layers, *BusinessUnit*, *Department*, and *Team*. A typical implementation will not reflect these domain concpts directly. Instead, there will probably be one implementation concept *OrgUnit* and the *Composite Pattern* [21] applied to it.

Figure 13 outlines a formalization of the pattern. A pattern defined this way can be applied by "instantiating" the pattern

```
PatternDefinitions {

  CompositePatternApplication {
    TargetModel
    ComponentClass
    CompositeClass
    LeafClass
    AggregationRelationship
  } |= TargetModel {
    ComponentClass is an AbstractClass
           from ObjectOrientedProgramming
    LeafClass      is a  Classifier
           from ObjectOrientedProgramming,
                      a  ComponentClass
    CompositeClass is a  Classifier
           from ObjectOrientedProgramming,
                      a  ComponentClass {
      AggregationRelationship
        is a ComponentClass
    }
  }
}
```

Figure 13. Layout of a pattern definition.

```
CompanyImpl {

  OrgEntity is an AbstractClass
           from ObjectOrientedProgramming

  Person    is a  ConcreteClass
           from ObjectOrientedProgramming,
               an OrgEntity

  OrgUnit   is a  Classifier
           from ObjectOrientedProgramming,
               an OrgEntity
  {
    Members is an OrgEntity
  }

}
```

Figure 14. Sample implementation resulting an application of the composite pattern.

concept, *CompositePatternApplication* in the example of the company organization as shown in Figure 12.

With the placeholders *TargetModel*, *ComponentClass*, *CompositeClass*, and *LeafClass* set by "is the", they each evaluate to the actual concepts given in the pattern application. Therefore, they will generate the model structure shown in Figure 14. Please note that the resulting implementation model *CompanyImpl* amends the definition in Figure 12, which leads to the concrete *OrgUnits* being available in the implementation, but now combinable via the *Members* relationship.

This resulting software model is not as accurate as the domain model, but better reflects the generalized way in which the model will be implemented.

VII. 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 article, we propose an approach for defining code generators within the MDSE toolchain based on models. Generic models lay a foundation for all code generators. To this end, we studied models of typical programming paradigms with certain language-specific properties. Code generators consist of executable models that are derived from the base models. They should, therefore, be formulated in the same modeling language. If the models that define a code generator are also 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. We use the M^{3}L as a consistent modeling framework.

The proposed approach allows us to achieve the goals of reduced development costs for code generators and of increased reuse, of both the base models and of parts of application-specific models. The use of multiple levels of abstraction makes each development step easier and less costly. Since the most abstract models can be applied in a generic way, they can be reused in different 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.

Acknowledgments

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

References

[1] H.-W. Sehring, "Building model-based code generators for lower development costs and higher reuse", in *Proceedings Nineteenth International Conference on Software Engineering Advances*, ThinkMind, 2024, pp. 26–31.

[2] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap", in *Future of Software Engineering (FOSE '07)*, 2007, pp. 37–54.

[3] K. Czarnecki and S. Helsen, "Classification of model transformation approaches", in *Proceedings OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, vol. 45, 2003, pp. 1–17.

[4] H.-W. Sehring, S. Bossung, and J. W. Schmidt, "Content is capricious: A case for dynamic system generation", in *Proceedings Advances in Databases and Information Systems*, Springer, 2006, pp. 430–445.

[5] H. Mannaert, K. D. Cock, and J. Faes, "Exploring the creation and added value of manufacturing control systems for software factories", in *Proceedings Eighteenth International Conference on Software Engineering Advances*, ThinkMind, 2023, pp. 14–19.

[6] J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio, "Dealing with the coupled evolution of metamodels and model-to-text transformations", in *Proceedings of the Workshop on Models and Evolution*, ser. CEUR Workshop Proceedings, vol. 1331, CEUR-WS.org, 2014, pp. 22–31.

[7] K. Lano and Q. Xue, "Code generation by example using symbolic machine learning", *SN Computer Science*, vol. 4, 2023.

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

[9] 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.

[10] K. Radek and J. Vladimír, "Incorporating Petri nets into DEVS formalism for precise system modeling", in *Proceeding Fourteenth International Conference on Software Engineering Advances*, ThinkMind, 2019, pp. 184–189.

[11] K. Czarnecki, "Overview of generative software development", in *Unconventional Programming Paradigms*, Springer Berlin Heidelberg, 2005, pp. 326–341.

[12] S. Trujillo, M. Azanza, and O. Diaz, "Generative metaprogramming", in *Proceedings of the 6th International Conference on Generative Programming and Component Engineering, GPCE '07*, Association for Computing Machinery, 2007, pp. 105–114.

[13] Object Management Group, *MOF model to text transformation language, v1.0*, [Online] Available from: https://www.omg.org/spec/MOFM2T/1.0/PDF. 2024.7.4. OMG Document Number formal/2008-01-16, 2008.

[14] M. Harter, "LLM Assisted No-code HMI Development for Safety-Critical Systems", ThinkMind, 2023, pp. 8–18.

[15] S. Hajiaghapour and N. Schlueter, "Evaluation of different systems engineering approaches as solutions to cross-lifecycle traceability problems in product development: A survey", in *Proceedings International Conference of Modern Systems Engineering Solutions*, ThinkMind, 2023, pp. 7–16.

[16] A. Agrawal, "Metamodel based model transformation language", in *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Association for Computing Machinery, 2003, pp. 386–387.

[17] D. Song, K. He, P. Liang, and W. Liu, "A formal language for model transformation specification", in *Proceedings of the Seventh International Conference on Enterprise Information Systems - Volume 3: ICEIS*, INSTICC, SciTePress, 2005, pp. 429–433.

[18] A. P. Fontes Magalhaes, A. M. Santos Andrade, and R. S. Pitangueira Maciel, "Model driven transformation development (mdtd): An approach for developing model to model transformation", *Information and Software Technology*, vol. 114, pp. 55–76, 2019.

[19] A. Kusel et al., "Reuse in model-to-model transformation languages: Are we there yet?", *Software & Systems Modeling*, vol. 14, pp. 537–572, 2 2015.

[20] D. Thomas and B. M. Barry, "Model driven development: The case for domain oriented programming", in *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Association for Computing Machinery, 2003, pp. 2–7.

[21] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994.