2 The State of the Art in Code Generation

Some of the requirements for the Genesys approach presented in Sect. 1.1 are a direct result of examining and evaluating the work that has been done in the field of code generation so far. This chapter provides an overview of the current state of the art in code generation for MD*. It starts off with a brief retrospect on classical compiler construction (Sect. 2.1), which developed ideas and concepts that clearly influence current code generation techniques. Sect. 2.2 elaborates on the conceptual foundations of MD* and on how the associated terminology is used in this book. Afterwards, Sect. 2.3 examines the role of code generation in several existing MD* (and related) approaches, and Sect. 2.4 introduces techniques for actually realizing code generators. Sect. 2.5 presents the state of the art in verifying and validating code generators. Finally, Sect. 2.6 compares Genesys with the approaches and techniques described in the preceding sections.

2.1 Influences of Compiler Construction

Beyond doubt, compiler construction is one of the most well-grounded and well-proven fields in computer science. Having its seeds in the early 1950s, compiler construction promoted the evolution of important theoretical topics such as formal languages, automata theory and program analysis. The introduction of compilers had far-reaching effects on software development, as they enabled the use of high-level programming languages (such as FORTRAN) instead of tediously writing software in low-level languages like assembly or even machine code. By *raising the level of abstraction*, developers should be shielded from hardware-specific details.

Code generation approaches for MD* share these ideas. According to Selic, "most standard techniques used in compiler construction can also be applied directly to model-based automatic code generation" [Sel03]. However, as models are by their very nature more abstract than source code (cf. Sect. 2.2), corresponding code generators work on a much higher level of abstraction than compilers for source code. The following paragraphs highlight some similarities as well as differences between classical compilers and MD* code generators, focusing on concepts and notions that are important for the Genesys approach.

General Structure:

In essence, a compiler translates a program written in a given source language into a program in a given target language. Usually, modern compilers are organized into consecutive phases, such as lexing, parsing or data flow analysis, each of them often operating on their own intermediate language or representation [App98, p. 4]. Depending on whether such a phase is concerned with analysis (i.e., resolving the source program into its constituent parts, assigning a grammatical structure, etc.) or synthesis (i.e., constructing the desired target program), the phase is said to be part of the compiler's front-end or back-end [Aho+06, p. 4], respectively. One of these phases is called "code generation", which is situated in a compiler's back-end. It usually retrieves some intermediate form, such as an abstract syntax tree produced by a parser, and translates it to code in the desired target language, e.g., machine code or bytecode executable by a virtual machine. This translation typically raises issues such as instruction selection, register allocation or code optimization.

For MD* code generators, especially issues close to hardware are at most secondary, and can often even be considered commodity. When generating code from abstract models, target languages are in most cases high-level languages (such as Java or C++) with existing compilers, interpreters or execution engines that further process the generated output. Accordingly, compilers can be regarded as tasks or services that are incorporated in or postpositioned to code generators. In a similar fashion, MD* code generators employ parsers in order to translate models from their serialized form (e.g., XML Metadata Interchange, XMI [Obj07]) to an in-memory representation (e.g., an implementation of the Java Metadata Interface, JMI [Jav02]) prior to the actual code generation. As there is extensive tool-support for the development of compilers and their single components, e.g., parser generators such as ANTLR [PQ95] or Lex/Yacc [LMB92], code generator developers can resort to a rich repertoire of mature services.

Bootstrapping:

Apart from source and target language, the compiler's implementation language is relevant to the categorization of the compiler. For instance, a *self-compiling* (or *self-hosting*) compiler [LPT78] is a compiler that is written in the language it compiles, and a *cross-compiler* [Hun90, p. 8] targets a machine other than the host. Especially self-compiling compilers are often used for *bootstrapping* [Wat93, p. 44], which is a common technique for evolving compilers. Typically, this approach aims at decreasing the overall complexity of compiler development by separating the implementation process into consecutive stages.

Fig. 2.1 uses the established notation of *T-diagrams* [Hun90, p. 11] for visualizing an example of a very simple bootstrapping process. In this notation, blocks that look like the letter "T" represent compilers. The three text labels on the blocks indicate the compiler's source language (left), target language (right) and implementation language (bottom). Suppose we want to implement a native compiler for a fictitious programming language called L. As a start, we implement version 1 of this compiler using C, an existing programming language with an available native compiler (M is for "machine code"). Afterwards, we compile the newly written compiler, which results in a native L-to-M compiler. We could stop at this point, but as the maintenance of our L-to-M compiler now depends on the existence of a C-compiler, we implement a second version in L (rebuilding should not be as hard as building from scratch). Finally, we compile version 2 using version 1 and get a native L-to-M compiler that is no longer dependent on C.

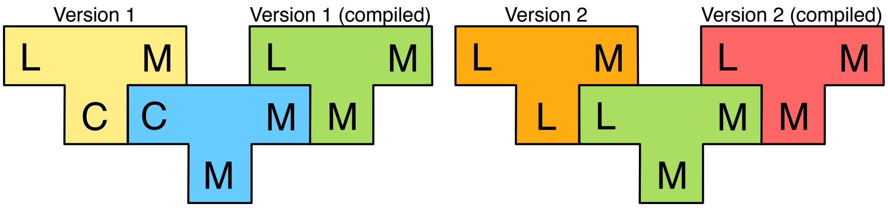


Fig. 2.1. Simple Bootstrapping Example: Getting a Native Compiler for Language L

The example in Fig. 2.1 is only a very small bootstrapping process. As mentioned above, bootstrapping is usually organized in stages in order to divide the implementation complexity into small manageable chunks. Instead of starting with the entire language L, a simple subset $L^* \subset L$ is identified, so that the first version of the compiler can be developed much easier. After building an L^*-to-M compiler in the manner described above, the compiler is enriched with the missing L-features and the procedure is repeated. Using several sublanguages with small feature additions in each stage further simplifies the implementation of the final compiler version.

The use of bootstrapping is also very common and desirable in MD* code generators, and thus an important technique used in the Genesys approach (see Sect. 1.1, 5.1 and 7.5). Throughout this work, T-diagrams will be used to visualize bootstrapping and other code generator evolution processes.

2.2 Models, Metamodels and Domain-Specific Languages

The existence of MD* approaches and numerous corresponding tools (cf. Sect. 2.3) indicates that there seems to be at least a common intuition of what a model actually is. However, there is still no generally accepted definition of the term “model”. For instance, while Kleppe defines a model as “a linguistic utterance of a modeling language” [Kle08, p. 187], the Object Management Group (OMG) focuses on the role of the model as a means of specification [Obj03b, p. 12]:

> *“A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language.”*

Kühne emphasizes the abstraction aspect of models [Küh06]:

> *“A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made.”*

Another characterization of models that is cited frequently in the literature is the one of Stachowiak, who identifies three main features of models [Sta73, pp. 131–133]:

1. *Mapping feature:* A model is always a mapping of some natural or artificial original, which may in turn be a model.
2. *Reduction feature:* Generally, a model does not capture all attributes of the represented original, but only those relevant to the person who creates or uses the model.
3. *Pragmatic feature:* A model always serves a particular purpose.

This “fuzziness” or lack of precision can be observed for most of the vocabulary used in the context of MD*. There is still no established fundamental theory of modeling and related concepts that would be comparable to the maturity achieved in other disciplines of computer science, such as compiler construction (cf. Sect. 2.1). However, several publications (e.g., [BG01;Fav04;Küh06]) try to come up with precise definitions, and thus discuss issues like when it is appropriate to call a model a *metamodel*.

As a reflection of this discussion goes far beyond the scope of this monograph, all following chapters and sections resort to the terminology definitions described by Stahl et al. [Sta+07, pp. 28–32]. Fig. 2.2 uses the Unified Modeling Language (UML) [Obj10b; Obj10a] in order to illustrate the relevant concepts and their relationships, which are introduced in the following.

Domain:

A *domain* is a delimited field of interest or knowledge which consists of “real” things and concepts. It may also be divided into an arbitrary number of *subdomains*. For instance, the domain “hospital” contains, among other things, the subdomains “intensive care unit” and “coronary care unit”, each capturing specific parts of the superordinate domain.

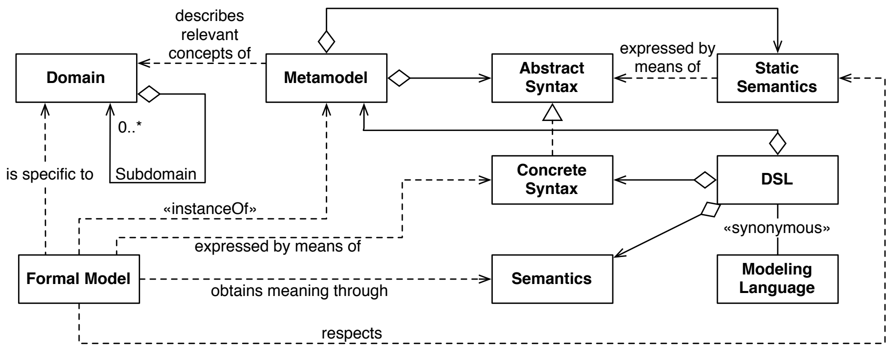


Fig. 2.2. Basic MD* terminology (by Stahl et al. [Sta+07, p. 28], translated into English)

Metamodel:

A *metamodel* is a formal description of a domain's relevant concepts. It specifies how formal models (or programs), that are specific to the given domain, can be composed. For this purpose, a metamodel comprises two important parts: the abstract syntax and the static semantics.

The *abstract syntax* defines the elements of the metamodel and their relationships, independent of the concrete representation of any corresponding formal model. For instance, the abstract syntax of an object-oriented language might define concepts like classes and interfaces, which have attributes such as a name and which are associated via relationships such as inheritance.

The *static semantics* specifies constraints for the well-formedness of a formal model. Accordingly, it is defined relative to an abstract syntax, i.e., it uses the contained terminology and concepts in order to describe the constraints. For instance, the static semantics of a metamodel for control flow graphs could specify constraints that demand the existence of exactly one start node.

Domain-Specific Language:

According to Fowler, the notion *domain-specific language* (DSL) refers to "a computer programming language of limited expressiveness focused on a particular domain" [Fow10, p. 27]. Stahl et al. [Sta+07] as well as this book use the notion synonymously with the term *modeling language*. As visible in Fig. 2.2, a DSL is based on a metamodel that comprises the abstract syntax and static semantics as described above.

Furthermore, a DSL provides a *concrete syntax*, which describes a particular representation of the elements and concepts specified by the abstract syntax. The concrete syntax can thus be considered an instance of the abstract syntax, and it is possible to define multiple concrete syntaxes for one abstract syntax. For instance, a UML class diagram [Obj10b] can be represented using at least three concrete syntaxes: the graphical UML notation itself, the Human-Usable Textual Notation (HUTN) [Obj04] and the XML-based interchange format XMI. In particular, this example illustrates that a concrete syntax – and thus the DSL and any formal model that follows the concrete syntax – can be textual or graphical.

The beginning of Chap. 1 presented several arguments that highlight advantages of graphical notations over purely textual notations (better cognitive accessibility, higher expressiveness, flatter learning curves etc.). However, there are also publications that argue in favor of textual notations. For instance, from the tool perspective, Völter [Völ09] points out that it requires more effort to build usable editors for graphical languages as opposed to textual editors. Stahl et al. [Sta+07, p. 103] exemplify this by means of the support for collaborative development: Whereas the synchronization of textual development artifacts is supported by a variety of tools (such as Subversion [Apa11e]), graphical notations often require the implementation of specific solutions.

Further positions advocate that graphical and textual notations are not mutually exclusive. Van Deursen et al. [DVW07] observe complementary strengths and thus propose a unification of both notations. Kleppe exemplifies UML class diagrams as such a hybrid concrete syntax, as they provide "a textual syntax embedded in a graphical one" [Kle08, p. 5]. Finally, Kelly and Pohjonen point out that the choice of a suitable concrete syntax "depends on the audience, the data's structure, and how users will work with the data" [KP09].

As the third component besides the metamodel and the concrete syntax, a DSL also provides *semantics* that assigns a meaning to any well-formed model written in the DSL. In practice, this semantics is often described by means of natural language as for instance performed in the UML specification [Obj10b]. However, in order to avoid the ambiguity and imprecision of natural languages, semantics can also be described formally, e.g., using a denotational [Sch86], operational [Plo81;Kah87], axiomatic [Hoa69] or translational approach [Kle08, p. 136f]. In the context of this book, the latter is most interesting: Following the translational approach, the semantics of a language is given by a translation into another language with well-known semantics. In MD*, such a translation can be provided by a model transformation, which may, e.g., be realized by a code generator. Sect. 2.3.5 elaborates on this role of code generation.

Fowler [Fow10, p. 15] distinguishes between internal and external DSLs. An *internal DSL* (also known as *embedded DSL*) forms a real subset of an existing (general-purpose) language, its "host language". It employs the syntactic constructs of the host language and maybe also parts of its available tooling support. Several languages like Lisp [McC60] or Ruby [FM08] support the creation of such internal DSLs. In contrast to this, an *external DSL* uses a separate custom syntax that is not directly derived from an existing host language. Consequently, with an external DSL it is usually not possible to resort to existing tools, so that, e.g., a specific parser for the language has to be implemented.

Formal Model:

The box labeled *formal model* in Fig. 2.2 represents a program or model written in a particular DSL. Consequently,

- it describes something from the domain for which the DSL is tailored,
- it is an instance of the metamodel contained in the DSL and in particular respects the metamodel's static semantics,
- it is written using the concrete syntax of the DSL, and
- its meaning is given by the DSL's semantics.

Due to its "formal" nature, such a model is a suitable basis for activities like verification, interpretation or code generation. For the sake of simplicity, this book uses the notion "model" in place of "formal model", implicitly including textual as well as graphical incarnations.

Metamodeling and Metalevels:

The notions and concepts depicted in Fig. 2.2 can be applied to arbitrary *metalevels*. For instance, considering "modeling" itself as a possible domain, one could create a "meta-DSL" for describing DSLs. Accordingly, when using the meta-DSL to specify a particular DSL *myDSL*, this new DSL is an instance of (i.e., a formal model conforming to) the metamodel given by the meta-DSL, or in other words: The meta-DSL provides the metamodel of *myDSL*. Continuing the example, *myDSL* can now in turn be used to create a particular model *M*, i.e., following the same argumentation as above, *myDSL* provides the metamodel of *M*. However, given the fact that *myDSL* itself is formally described by means of the meta-DSL, the meta-DSL provides the *metametamodel* of *M*. Thus the role of the meta-DSL is determined relative to the metalevel from which it is observed. Accordingly, the "metaness" of a model arises from its relations to other models (being its instances) rather than being an intrinsic model property [Sta+07, p. 63].

A well-known example of a metamodeling architecture which employs metalevels is the Model Driven Architecture (MDA) [Obj03b] (cf. Sect. 2.3.3) proposed by the OMG. MDA enables model-driven software development on the technological basis of standards that are also created by the OMG, such as the Meta-Object Facility (MOF) [Obj11d] and UML. Fig. 2.3 is a slightly extended version of an illustration from [Obj10a, p. 19], showing an example of metalevels in MDA. The single metalevels are typically labeled M0, M1, M2 and so on, with M0 designating the lowest level. M0 usually represents the actual system (existing or non-existing) that is to be modeled, or more precisely its runtime objects and user data. The models that represent this system are situated on level M1, e.g., concrete diagrams (class diagrams etc.)

modeled in UML. Level M2 holds the modeling language that is used for describing the models on M1, i.e., their metamodel. For instance, in the MDA context, this might be the UML along with its associated concepts. Finally, the metamodel on M2 is again formally described by a model which is situated on level M3, the metametamodel. In MDA, this role is played by MOF, and thus in order to be MDA-compliant, a modeling language has to be an instance of (i.e., it has to conform to) MOF. Please note that only levels M1–M3 (and maybe above) are actual modeling levels, as M0 represents the "real" system (which is why, e.g., Bézivin refers to the four-level example as a "3+1 architecture" [Béz05]).

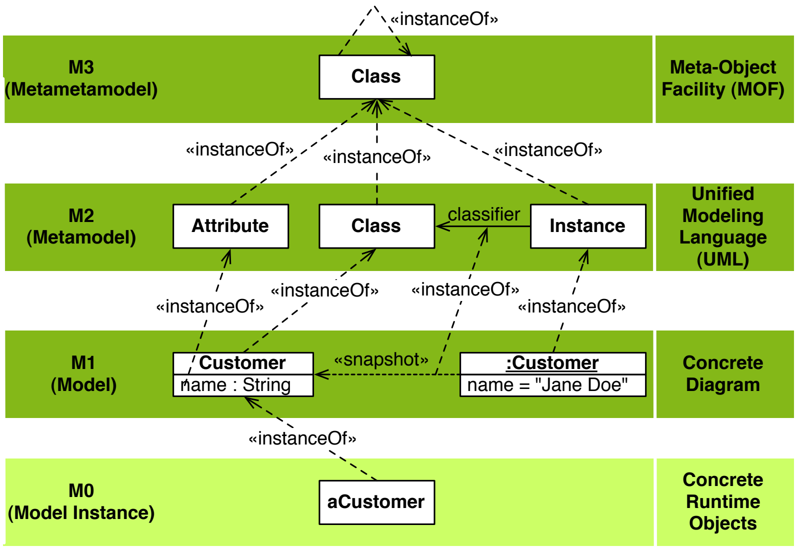


Fig. 2.3. Four-Level Example of MDA's Metamodel Hierarchy (based on [Obj10a, p. 19])

Except for the topmost metalevel, the elements of each level are instances of elements in the level above. Conceptually, there is no need for such a "hierarchy top" at all – the number of metalevels can be arbitrary [Obj10a, p. 19]. However, in practice, this potentially indefinite layering is usually avoided by means of a *reflexive* model, i.e., a model that is able to describe itself [Sei03; Sel09]. As indicated in Fig. 2.3, MOF is such a model that is defined in terms of itself, so that effectively no more metalevels are required. Another example of a reflexive model is *Ecore* from EMF (see Chap. 7).

It should be noted that the one-dimensional view on metalevels shown in Fig. 2.3 is subject to controversy. For instance, Atkinson and Kühne [AK02] pointed out that it fails to distinguish different types of "instance of" relationships and thus proposed a two-dimensional framework. However, a detailed discussion of those issues goes beyond the scope of this book.

Users of MD* tools usually only deal with a restricted view on the available metalevels. For instance, in typical UML tools like ArgoUML [Tig11], any modeling activity happens exclusively on level M1, i.e., the levels M2 and M3 are "hard-wired". Other tools such as language workbenches (see Sect. 2.3.5) also allow the user to define his own modeling languages and thus hard-wire only level M3 or above.

2.3 The Role of Code Generation

As pointed out in Chap. 1, code generation is key to any MD* approach to software development. It bridges the gap that arises when models are used to abstract from the technical details of a concrete software system. Code generation is thus an enabling factor for allowing real *model-driven* software development which treats models as primary development artifacts [Sei03; Béz05], as opposed to the approach termed *model-based* software development in Chap. 1 that is limited to using models for documentation purposes [Sta+07, p. 3].

Apart from the notion *MD**, which is used in this book (following Völter [Völ09]) as a generic term for referring to the variety of existing approaches to model-driven development, there are several further notions that are used in a similar way. Examples that can be frequently found in publications are Model-Driven Development (MDD) [Sel03; AK03], Model-Driven Engineering (MDE) [Sch06; Béz05; Fav04; DVW07] and Model-Driven Software Development (MDSD) [Sta+07], which are largely used synonymously. Among MD* approaches, code generation is usually considered a specific form of model transformation and thus often referred to as *model-to-text transformation* [CH06; Old+05] or *model-to-code transformation* [Sel03; Sta+07; Hem+10].

The following sections (2.3.1–2.3.5) provide examples of existing MD* and related approaches, with a particular focus on the respective role of code generation. Afterwards, Sect. 2.3.6 briefly sketches MD* approaches that do not resort to code generation.

2.3.1 Computer-Aided Software Engineering

The idea of automatically generating an implementation from high-level specifications is not really new. For instance, in the 1980s, the *Computer-Aided Software Engineering* (CASE) approach [CNW89] had very similar objectives, including the design of software systems by means of graphical general-purpose languages and the use of code generators for automatically producing suitable implementations [Sch06].

However, the CASE approach has not asserted itself in practice. As one reason for this, Schmidt [Sch06] especially designates the deficient translation of

CASE's graphical general-purpose languages to code for desired target platforms. The creation of corresponding code generators was very difficult as the produced code had to compensate the lack of important features, such as fault tolerance or security, in operating systems at that time. As a result, the code generators were very complex and thus hard to maintain. Moreover, CASE tools focused on proprietary execution environments, which resulted in low reusability and integrability of the generated code. Schmidt also names further problems of CASE, such as the lack of support for collaborative development and the fact that the employed graphical languages were too generic and too static to be applicable in a large variety of domains. Especially as a result of the insufficient code generation facilities, CASE tools were often used for model-based software development only [Sch06].

Today's MD* approaches benefit from the fact that programming languages and platforms significantly evolved since that time. Apart from the fact that code generation technologies have matured [Sel03], code generation has become much more feasible, as generators "can synthesize artifacts that map onto higher-level, often standardized, middleware platform APIs and frameworks, rather than lower-level OS APIs" [Sch06], which decreases their complexity significantly.

Moreover, as another lesson learned from CASE, lots of MD* approaches advocate the use of DSLs rather than general-purpose languages, thus turning away from CASE's "one size fits all" idea [Sta+07, p. 44]. The focus on DSLs further increases the significance of code generation, as the specification of a DSL often entails the demand for a corresponding code generator – or multiple ones if several target platforms are used –, the creation of which also needs to be supported by appropriate frameworks and tools.

2.3.2 Generative Programming

Generative Programming (GP) [CE00], also known as *Generative Software Development*, is an approach that "aims at modeling and implementing system families in such a way that a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific languages" [Cza04].

Accordingly, it puts particular emphasis on two main aspects. First, GP focuses on developing families of systems instead of only single systems. A system family is a set of systems based on a common set of assets [CE00, p. 31], which are used for building the single family members. Among other things, such a system family might form the basis for creating product lines [Sta+07, p. 35]. Second, GP involves the automatic assembly of the final system via generators. Inspired by industrial manufacturing, the generated system should resemble a complete, "highly customized and optimized intermediate or end-product" [CE00, p. 5].

The common model that is used for generating the single members of a system family is called the *generative domain model*. This model essentially describes three components: the problem space, the solution space as well as a mapping between both. The *problem space* can be considered the domain, and it contains one or more domain-specific languages that provide the concepts and terminology for specifying system family members. For instance, *feature models* [CHE04] are frequently used in connection with GP as a means for describing the common features of system family members along with those features that are variable. Feature models also capture how variable features depend on each other. The *solution space* consists of elementary implementation components which are used to assemble a system. The mapping between problem space and solution space is given by *configuration knowledge*, which includes illegal combinations of features, default settings and dependencies as well as construction rules and combinations [CE00, p. 6]. This configuration knowledge is implemented by means of one or more generators.

Based on this generative domain model, a system is essentially specified via configuration: An application programmer creates such a configuration by selecting desired features in the problem space, and the generator uses the configuration knowledge for automatically mapping it to a configuration of implementation components in the solution space. Besides this *configuration view* [Cza04] further describes a *transformational view* on the generative domain model. In this view, the problem space is resembled by a domain-specific language which is transformed into an implementation language situated in the solution space. Independent of the particular view, GP does not dictate which technologies are used for actually implementing the single elements of the generative domain model [CØV02].

GP is strongly related to MD* approaches as both advocate the use of DSLs for creating high-level specifications along with corresponding generators that automatically produce a system from those specifications. However, GP's strong focus on the development of software system families distinguishes it from several MD* approaches such as MDA (see the following section). Whereas MDA mainly addresses technical variability by aiming at portability, GP also takes application domain variability into account [Cza04]. Furthermore, Stahl et al. [Sta+07, p. 39] point out that GP traditionally focuses more on textual DSLs rather than on graphical notations.

In particular, lots of research in the realm of software product line engineering [CN01; PBL05] relates to GP's mindset. A recent example is the HATS project [Cla+11], which employs Abstract Behavioral Specification (ABS) in order to model system families. To this end, ABS consists of five textual languages for specifying

1. core modules of the system in a behavioral fashion,
2. the system's features and their attributes via feature modeling,
3. variability of the system by means of delta modeling [Sch+10],
4. product line configurations that link features with delta modules, and
5. concrete product selections.

From the GP perspective, those specifications provide the required concepts in the problem space as well as the configuration knowledge required for the mapping into the solution space. Finally, a concrete product is generated via a dedicated compiler, which, for instance, is able to translate an ABS model into Java code.

2.3.3 Model Driven Architecture

As mentioned in Sect. 2.2, *Model Driven Architecture* (MDA) [Obj03b] is an initiative of the OMG. It has been introduced in 2001 and primarily aims at "portability, interoperability and reusability through architectural separation of concerns" [Obj03b, p. 12]. Conceptually, MDA defines three models that represent different viewpoints on a system:

- *Computation-Independent Model* (CIM): Also termed "domain model" or "business model" [Fra02, p. 192], the CIM describes the pure business functionality including the requirements of and rules for the system. Any technical aspects of the system are ignored. CIMs are supposed to be created and used by business experts (or "domain practitioners" [Obj03b, p. 15]) and thus use familiar terminology of the respective domain. They are intended as a bridge between business experts who are versed with a particular domain, and IT experts who have the technical knowledge for realizing a system. CIMs provide a very broad view as they also may contain aspects of a domain that are not automated at all [Fra02, p. 194].
- *Platform-Independent Model* (PIM): In contrast to CIMs, PIMs also consider technical aspects of a system, but only those which are independent of a concrete platform. This platform-independence is key to achieving the goal of portability, however it should be noted that it is a relative notion. Frankel [Fra02, p. 48f] exemplifies this by means of OMG's middleware standard, the Common Object Request Broker Architecture (CORBA) [Obj11b], which can be considered platform-independent as it does not depend on particular programming languages or operating systems. However, when viewing CORBA as one among many existing middleware technologies, it also can be considered a specific platform. From this perspective, platform-independence is only achieved by not depending on a concrete middleware technology. Accordingly, a PIM "exhibits a specified degree of platform-independence so as to be suitable for use with a number of different platforms of similar type" [Obj03b, p. 16].
- *Platform-Specific Model* (PSM): A PSM augments a PIM by further technical details that are specific to a particular platform. Please note that the above comments on the relativity of platform-independence can be similarly applied to platform-specificity.

Further OMG standards provide the technological basis for creating such models: Any modeling language that conforms to MOF (see Sect. 2.2) can be used, such as UML or the Common Warehouse Metamodel (CWM) [Obj03a].

PIMs, PSMs and the actual implementation code of the system are connected by means of *transformations*. For instance, a PIM could be successively refined by one or several consecutive model transformations producing either further PIMs or PSMs, the last of which being the most concrete or specific model that is used as the basis of a final code generation step. However, the creation of intermediate models is not mandatory, as it might also be possible (e.g., depending on the abstractness of the employed PIM) to produce code directly from a PIM [Obj03b, p. 25]. The exact nature of the transformation is not dictated by MDA: A transformation may, e.g., be entirely manual, semi-automatic by marking the models with additional information, or fully automatic [Obj03b, pp. 34–36].

Model transformations (PIM to PIM, PIM to PSM, PSM to PSM) can, e.g., be realized by using any implementation of OMG's Query/View/Transformation (QVT) [Obj11c] specification. Another example for a language that supports such model transformations is the Atlas Transformation Language (ATL) [JK06]. Both QVT and ATL are, e.g., implemented in the context of the Model 2 Model (M2M) project which is part of the Eclipse Modeling Project (EMP) [Gro09].

For code generation (PIM to code, PSM to code), there exists a plethora of tools and frameworks such as AndroMDA (which has been used for a case study in the context of this monograph and thus will be described in more detail in Sect 8.1), MOFScript [Old+05], Fujaba [GSR05] or XCoder [Car11]. Moreover, there are implementations of OMG's MOF Model to Text Transformation Language (MOFM2T) [Obj08] like Acceleo [Obe11], and integrated code generation facilities in tools that support UML modeling, such as Altova UModel [Alt11] and Together [Bor11].

Although the MDA has gained lots of attention and is, in the author's assessment, perhaps the most widely known MD* approach, some of its related standards are subject to criticism. For instance, Sect. 2.2 already pointed out that the one-dimensional metamodeling architecture specified by MOF was controversial – however, the situation improved significantly with the introduction of UML 2.0 and MOF 2.0 (though still some issues remain [AK03]).

Maybe the most contentious part of MDA is UML. A major point of criticism is its lack of a clearly and formally described semantics [Tho04; BC11]. Furthermore, Kelly and Tolvanen point out the low abstraction provided by UML models, which "are at substantially the same level of abstraction as the programming languages supported" [KT08, p. 19f], because "the modeling constructs originate from the code constructs" [KT08, p. 14] instead of deriving them from the domain of the modeled system. Another problem arises from the practical difficulty of synchronizing the various UML models that describe different aspects of a system: When changes to a model are not propagated to dependent models, this may lead to inconsistencies that hamper the system's evolution [Hör+08]. In particular, this issue also concerns *round-tripping*, i.e., the synchronization of UML models and the code generated from them – Sect. 2.4.4 further elaborates on this.

2.3.4 Domain-Specific Modeling

Domain-Specific Modeling (DSM) [KT08] explicitly focuses on the creation of solutions that are entirely tailored to a particular domain. According to Kelly and Tolvanen, DSM typically includes three components: a domain-specific modeling language, a domain-specific code generator and a domain framework [KT08, p. xiii f]. Once those components are in place, developers use the domain-specific modeling language for creating models which are automatically translated into code. The use of the term "domain-specific *modeling* language" (instead of just DSL) can be considered to reflect a tendency of DSM towards visual notations "such as graphical diagrams, matrices and tables" [KT08, p. 50], that are used along with text (i.e., hybrid concrete syntaxes as described in Sect. 2.2). Furthermore, DSM clearly aims at *full code generation* (cf. Sect. 2.4.4), so that the generated code is complete and does not have to be touched [KT08, p. 49f]. In order to reduce the complexity of code generators, the produced code often is executed on top of a dedicated domain framework. Such a domain framework provides elementary implementations that do not have to be generated and thus relieve and simplify the code generator.

Kelly and Tolvanen point out that full code generation is achievable, because the language and the generator employed in DSM "need [to] fit the requirements of only one company and domain" [KT08, p. 3], thus strictly following the tenet that "Customized [sic] solutions fit better than generic ones" [KT08, p. xiv]. As a consequence of this orientation, DSM typically does not involve shipping of ready-made DSLs or code generators, because both are developed in-house as a part of implementing a DSM solution for a particular domain. In [TK09], Tolvanen and Kelly state that based on their industry experiences, this implementation phase is usually very short, with the time required for implementing the generator often outweighing the time for realizing the language.

In order to enable this modus operandi, proper tooling is required that supports both the definition and the usage of a DSM environment for creating a particular domain-specific solution. Consequently, tools for DSM usually have a hard-wired metametamodel (i.e., level M3, see Sect. 2.2), thus allowing the definition of new metamodels, ergo new domain-specific modeling languages. In this respect, DSM tools contrast with CASE or UML tools [KT08, p. 60], which usually dictate the use of a particular modeling language.

Perhaps the most prominent DSM tool is MetaEdit+ [TK09; KLR96]. As further tools that can be considered realizations of the approach, Kelly and Tolvanen [KT08, p. 390–396] mention the Generic Modeling Environment (GME) [Led+01] (originally developed in the context of Model-Integrated Computing [SK97]), Microsoft's DSL Tools [Coo+07] (a part of the Software Factories [Gre+04] initiative) and the EMF-based Graphical Modeling Framework (GMF) [Ecl11a; Gro09].

2.3.5 Language Workbenches

In 2005, Martin Fowler coined the term *language workbench* [Fow05] for referring to a class of tools that specifically focus on DSLs. This is not restricted to providing an IDE for creating a DSL (e.g., features for creating a metamodel or generating a parser): Language workbenches also support building a specialized IDE that is equipped with, e.g., custom editors and views for using the created DSL. Consequently, similar to tools for DSM mentioned in Sect. 2.3.4, language workbenches significantly differ from CASE and UML tools, which usually are based on a fixed metamodel [KT08, p. 60]. Altogether, a language workbench enables the definition of a DSL environment by specifying the metamodel, an editing environment and the semantics of the DSL ([Fow10, p. 130], adapted to the terminology introduced in Sect. 2.2).

For the custom editing environment, language workbenches usually employ either source editing or projectional editing [Fow10, p. 136]. *Source editing* uses one single representation for editing and for storing, which is usually text. The creation of such text does not depend on a particular tool but can be performed with any text editor. In contrast to this, with *projectional editing* the primary representation of a program or model is specified and tightly coupled with the employed tool. The tool provides the user with an editable projection of this representation, which might follow any concrete syntax (textual or graphical). Editing the projection then directly modifies the primary representation. In consequence, in this scenario, the user never works directly with the primary representation, and the tool is imperatively required for editing, as it has to perform the projection.

Projectional editing provides several advantages over direct source editing, such as the possibility to provide multiple (e.g., user-specific) projected representations. Graphical modeling tools naturally employ projectional editing, as the actual model is usually kept separate from its graphical representation. Thus the differentiation makes most sense for textual DSLs. Language workbenches that are based on projectional editing are also termed *projectional language workbenches* (see, e.g., [VV10]).

Code generation plays a central role for most language workbenches as it is frequently used for providing the semantics of a created DSL. According to Fowler, the semantics of the DSL is most commonly specified in a translational way (cf. Sect. 2.2), i.e., by means of code generation, and more rarely on the basis of interpretation [Fow10, p. 130]. Consequently, most workbenches provide means for specifying code generators, some of which will be exemplified in Sect. 2.4.

The rationale behind language workbenches is often associated with language-oriented programming (see, e.g., [Fow05; **?**]). The term has been coined by Ward [War94] in 1994 and refers to the general approach of solving a problem with one or more domain-specific languages rather than with general-purpose languages.

Many existing tools meet the characteristics of language workbenches described above. For instance, MetaEdit+ (presented in Sect. 2.3.4) can be considered a language workbench which supports the creation of graphical (or visual) DSLs along with projectional editing. Other language workbenches mainly focus on textual DSLs, providing either projectional editing like the Meta Programming System (MPS) [Jet11] or parser-based source editing like Xtext [Ecl11h], Spoofax [KV10] or Rascal [KSV09].

2.3.6 Approaches without Code Generation

For the sake of completeness, it should be noted that code generation is not the only way to obtain a running system from a model. Another common solution is the use of an interpreter which directly executes a model without previous translation.

Business Process Modeling (BPM) is an example of a field which predominantly employs model execution. Such models are usually business processes that are described by means of dedicated languages, and that are typically executed (i.e., interpreted) by a process engine. Examples are Business Model & Notation (BPMN) [Obj11a] with corresponding process engines like jBPM [Red11b] or Activiti [Act11b], and the Business Process Execution Language (BPEL) [OAS07] which can be executed by engines such as ActiveVOS [Act11a] or Apache ODE [Apa11c]. Typically, process engines provide features like scalability, long-running transactions (e.g., via persistency of process instances), support for human interactions and monitoring of running processes.

A major feature of interpreters is late binding. In BPM this is used, among other things, for running multiple versions of a process. It also allows, e.g., the realization of multi-tenancy capabilities, or of process adaptations at runtime. The latter is also a major goal of the "models@run.time" approach [BBF09] which aims at exploiting the advantages of models not just for software development, but also in the running system. For instance, models can be useful at runtime for realizing (self-)adaptive software systems.

Furthermore, an interpreter may play the role of a reference implementation that specifies the semantics of a DSL, as an alternative to describing the semantics in a formal way (cf. Sect. 2.2). Kleppe [Kle08, p. 135] refers to this as *pragmatic semantics.*

The choice between code generation and interpretation is not exclusive, as both approaches can be combined. For instance, the execution of generated Java code can be considered such a combination, as the Java Virtual Machine (JVM) [LY99] can be regarded an interpreter for bytecode. This book will show several further combinations of code generation and interpretation, such as interpreter-based bootstrapping of a code generator (Sect. 5.1) and the use of an interpreter via API in order to realize the execution of generated code (Sect. 5.1.1).

2.4 Code Generation Techniques

Similar to a compiler, a code generator can be characterized as a “T-shape” in a T-diagram (cf. Sect. 2.1): It supports a particular source language, translates to a desired target language and is implemented using a specific implementation language. Each of these three facets may be based on a different language. While the source and the target language are usually given by initial requirements, the implementation language has to be selected advisedly. For instance, it may be advantageous to use the same language as source and implementation language in order to enable bootstrapping (cf. Sect. 2.1).

Apart from the selection of an appropriate implementation language, there are also several approaches for the actual implementation of a code generator. Generally, each approach covers two aspects of the code generator. First, the *output description* specifies the structure and the appearance of the generated code. Second, the *generation logic* describes the logic of the code generator, i.e., how the mapping from the source language to the target language is actually performed. This may also include further actions such as pretty-printing, assembling code fragments or writing the code to corresponding files.

In the literature, different classifications are used for categorizing the existing approaches to code generation. For instance, Kleppe [Kle08, pp. 151–156] makes the following interrelated distinctions:

1. *Model transformation rules versus hard-coded transformation:* In the first case, the code generator is described by means of a set of transformation rules. These rules are processed by a corresponding tool which performs the actual translation from source to target language, and which thus realizes a large part of the generation logic via a generic transformation engine. In the second case, the transformation is implemented explicitly, e.g., using an imperative language.
2. *Source-driven versus target-driven transformation:* With source-driven transformation, the structure of the input model in the source language drives the code generation: The generator processes the input model and produces corresponding code in the target language for each model element. For instance, this might result in a set of code fragments that are assembled in a final step. If the translation is target-driven, the code generator is oriented towards the structure of the desired output. In such an approach, the code is, e.g., generated sequentially into some kind of stream, and each time any information from the input model is required, the model is specifically queried for it.
3. *Concrete form versus abstract form target:* A code generator may either translate into the concrete syntax of the target language or into a representation of its abstract syntax. Accordingly, in the latter case, the result is again a model resembling an abstract form of the code (see Sect. 2.4.3 for more details on this).

Czarnecki and Helsen [CH06] employ a much more coarse-grained and technical categorization as they only distinguish *visitor-based* and *template-based* approaches. The former use a form of the well-known visitor design pattern [Gam+95, pp. 331ff] for realizing the traversal of the input model and for mapping elements of the source language to elements of the target language (see also [Kle08, pp. 158f]). The approaches associated with the second category describe the code generation by means of templates, a combination of static text (i.e., the output description) and dynamic portions (which realize parts of the generation logic). In order to produce the actual code, a template engine evaluates the dynamic portions on the basis of the input model (see Sect. 2.4.2 for more details).

Fowler [Fow10, p. 124] also introduces two categories, called *transformer generation* and *templated generation*. Basically, templated generation equals Czarnecki and Helsen's category of templated-based approaches. With transformer generation, Fowler refers to any approach that processes the input model and emits code in the target language for each model element.

The following sections describe different techniques for realizing code generators and, where applicable and useful, assign them to the different categories outlined above. Finally, Sect. 2.4.4 elaborates on different types of outputs that can be produced with code generation.

2.4.1 Programming the Code Generator

The most minimalistic way to implement a code generator is to write it using a general-purpose programming language. As in this case the transformation from source language to target language is explicitly implemented, the resulting code generators belong to Kleppe's "hard-coded transformation" category. In the sense of Fowler's classification, those generators are an application of transformer generation.

Implementing a code generator this way only requires an API for accessing the models programmatically. The actual output is typically assembled by means of basic string concatenation. Accordingly, output description and generator logic are usually mixed up in such implementations. Moreover, depending on the selected programming language, the required handling of strings may increase the complexity of the implementation: If, e.g., Java is selected as the implementation language, special characters (such as quotation marks) have to be escaped and explicit operators (e.g., +) have to be employed for the concatenation of strings [Sta+07, pp. 150f].

In parts, this complexity can be hidden by means of dedicated code generation APIs. As described by Völter [V03], such an API is designed to resemble the abstract concepts of the target language. For instance, if Java is the target language, a corresponding code generation API would provide concepts like classes, methods, modifiers etc. as manipulable objects. After manipulation, each of those objects would be able to produce its own code in the target language. Consequently, the generator developer only has to deal with the API, which relieves him of tedious tasks such as low-level string concatenation.

Additionally, the visitor pattern (see above) can be applied for realizing the mapping of the API objects to corresponding code non-invasively and at a central place. Czarnecki and Helsen [CH06] mention the code generator framework Jamda [Boo03] as an example of an API- and visitor-based approach.

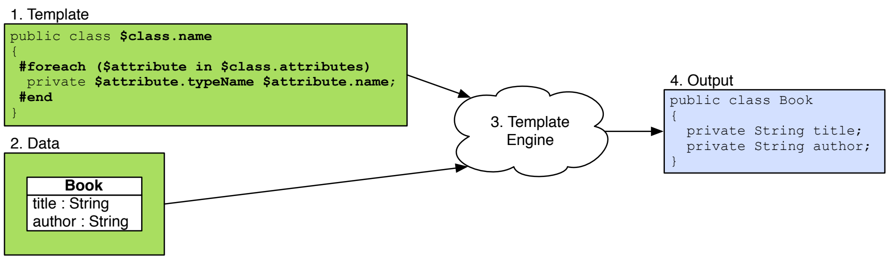
Code generators which are implemented "per pedes" based on a general-purpose programming language and APIs are usually sufficient for small application scenarios, which do not require generating a large amount of complex code. However, for larger scenarios such code generators usually do not scale well as in this case they are much harder to write [V03] and to maintain. Furthermore, Kelly and Tolvanen [KT08, p. 271] point out that many general-purpose languages do not provide convenient support for the navigation of complex models and the production of text at once.

A possible solution to the latter problem is the selection of a programming language which provides facilities that are specifically designed to support the implementation of code generators. An example of such a language is Xtend 2 [Ecl11g] which is used in recent versions of Xtext (version 2 at the time of writing this text). As another solution, Kelly and Tolvanen propose the use of a dedicated DSL, which allows a more concise description of a code generator than a general-purpose language. Furthermore, a DSL enables the specification of the code generator on a higher level of abstraction, thus hiding low-level issues. An example of such a DSL is MERL [KT08, p. 273] which is used for creating code generators in MetaEdit+. As a disadvantage of this solution, it is not possible to resort to existing tool support, which is typically readily available for general-purpose languages. Consequently, if the DSL is not an internal DSL, the implementation of specific tools for, e.g., executing and debugging the code generator may be required. For MERL, MetaEdit+ provides corresponding tools [TK09].

2.4.2 Template-Based Code Generation

This technique is based on the use of *templates*. Similar to a form letter [Sta+07, p. 146], a template consists of static text with embedded dynamic portions that are evaluated by a *template engine*. This approach is especially common in web development, where it is used by techniques such as Active Server Pages .NET (ASP.NET) [Mic11] or JavaServer Pages (JSP) [Jav09b] for dynamic server-side generation of web site contents.

Fig. 2.4 shows an example of a template and the general modus operandi of the approach. It is visible that apart from the actual template, a template engine also requires concrete data as an input. In order to generate the actual output, the dynamic portions of the template are evaluated on the basis of this data and replaced by corresponding static text.



```
public class $class.name
{
 #foreach ($attribute in $class.attributes)
  private $attribute.typeName $attribute.name;
 #end
}
```

```
public class Book
{
  private String title;
  private String author;
}
```

Fig. 2.4. Using a template engine for code generation

The template depicted in Fig. 2.4 describes the translation of a class noted as a UML class diagram into corresponding Java code. The dynamic portions of the template (visualized in bold face) are written in a *template language*. Such languages typically use dedicated control characters (in the example `$` and `#`) for distinguishing static from dynamic contents. In the example, it is visible that the template accesses the elements of the class diagram via the diagram's abstract syntax (defined in the corresponding metamodel). For instance, a class contained in the diagram is referenced by means of the expression `$class`, and also the properties of the class are accessed via suitable expressions such as `$class.name` or `$class.attributes`. Moreover, apart from such facilities for data access, most template languages support the use of control flow statements like conditionals, loops as well as method- or macro-calls. The example in Fig. 2.4 shows a `foreach` loop which iterates over all attributes of a class. For each attribute, the template describes the generation of a private member variable in the resulting Java class.

There is a large number of ready-made template engines which can be used for implementing a template-based code generator, such as StringTemplate [Par04], Velocity [Apa10], FreeMarker [Fre11b], Xpand [Ecl11f] and JET [Ecl11d]. Usually each template engine defines its own template language. For some template engines there is also sophisticated IDE support. For instance, Xpand is supported by an Eclipse-based editor that provides features such as syntax highlighting and code completion.

Template-based code generators are very common [KT08, p. 272], which can also be witnessed by the fact that Fowler as well as Czarnecki and Helsen consider them a category of their own. Examples of tools which employ template-based code generation are ANTLR (StringTemplate), EMF (JET), AndroMDA (Velocity, FreeMarker), Fujaba (Velocity), Acceleo (own template language) and former versions of Xtext (Xpand).

Similar to code generator implementation by means of a programming language (as described in Sect. 2.4.1), templates mix generation logic and output description. However, with a template-based approach, the generator developer is not confronted with issues such as escaping and string concatenation. Especially the latter is specified implicitly in the template and performed automatically and transparently by the template engine. As the structure of a template follows the structure of the output, the transformation is, in Kleppe's terminology, target-driven. Furthermore, template-based approaches belong to the category of hard-coded transformations [Kle08, p. 151].

Kelly and Tolvanen [KT08, p. 273] point out that working with templates can be inefficient if the generated output is distributed among multiple files (or locations). As a template usually resembles one file, a separate template is required for each output file and all templates have to be evaluated sequentially in order to produce the entire set of resulting files. This may lead to unnecessarily frequent traversals of the input model, even if information that is relevant for multiple files is located at the same place in the model.

2.4.3 Rule-Based Transformation

As mentioned above in Kleppe's categories, an alternative to hard-coding the transformation performed by a code generator is the use of *transformation rules*. In this approach, a set of such rules describes how each element in the source language is translated to a corresponding element in the target language. For the actual transformation, a *transformation engine* processes those rules and applies them to the input model given in the source language.

A code generator can be realized as a chain of such transformations. For instance, according to the MDA approach (cf. Sect. 2.3.3), such a chain is a sequence of model-to-model transformations on several intermediate representations, eventually ending with a final model-to-text transformation. Essentially, this idea is based on the classical "divide-and-conquer" paradigm: A complex transformation is handled by dividing it into smaller, simpler and thus more manageable steps.

Furthermore, approaches using chains of rule-based transformations often aim at an abstract form of the target language rather than at its concrete syntax (see Kleppe's "abstract form target" category). Instead of directly translating the original input model or any of the intermediate representations along the transformation chain to the concrete syntax of the target language, a structured representation (i.e., a model) of the target language is produced. The actual code is then produced by means of a final abstract-form-to-concrete-form transformation within the target language [Kle08, p. 155]. As the major advantage of targeting an abstract form, the abstract representation of the code is still available after the code generation. Thus it can be used for further processing steps and transformations, e.g., for extending the target language with additional constructs [Hem+10].

An example of rule-based transformations is described by Hemel et al. in [Hem+10]. They use Stratego/XT [Bra+08] (also employed by the language workbench Spoofax) for specifying the code generation via rewrite rules in combination with strategies for applying those rules. Another example is the language workbench MPS, which also allows rule-based transformation with abstract form target.

2.4.4 Round-Trip Engineering versus Full Code Generation

With regard to their results, code generators can be distinguished by means of two further categories: those which produce complete code and those which only generate stubs or skeletons that have to be completed by a developer.

Round-Trip Engineering:

Due to the fact that in the latter case models and code are both editable development artifacts, it is required to keep them mutually consistent. Performing this by hand is error-prone and increases the workload, because the same information has to be maintained at multiple locations. Consequently, a technique called *round-trip engineering* (RTE) [HLR08;SMW10] (also called *round-tripping* [KT08, p. 5]) aims at automating the synchronization between models and code. The both directions of this synchronization are also referred to as *forward engineering* (higher level model to lower level model or code) and *reverse engineering* (lower level model or code to higher level model) [MER99]. Accordingly, code generation belongs to the forward engineering techniques.

However, RTE has several problematic aspects. For instance, the forward engineering part has to ensure that the code can be regenerated safely when the model has been modified. This task is not trivial, especially when the code also has been subject to modification: In order to protect the developer's work, such changes must not be overwritten or invalidated by the regeneration.

According to Frankel, one possible solution is *partial round-trip engineering* [Fra02, p. 233–235], which restricts the allowed code modifications to additive changes. In this scenario, it is not allowed to overwrite or delete any code that has been generated from the model. At the same time, it is forbidden to add any code that could have been generated from a corresponding description in the modeling language. Consequently, the developer and the code generator only touch code for which they are exclusively responsible. This form of RTE is partial because it is unidirectional only – it does not support iterative reverse engineering [Fra02, p. 234].

Protected regions [KT08, p. 295f; Fra02, p.234]are a means for supporting such strictly additive code changes. Those regions are specific parts of the code that are, e.g., marked with dedicated comments. As a general rule, the developer must not perform any modifications outside of the protected regions. In turn, the code generator is able to detect the protected regions and leaves them untouched in case of a regeneration. However, as this feature needs to be supported by the code generator, this inevitably increases the complexity of the generator's implementation. Further problems with protected regions include modifications to the model which lead to the invalidation of manually written code (such as renaming of classes, methods etc.) [KT08, p. 66], or developers who do not stick to the rules and perform modifications outside of the protected regions [Fra02, p. 234].

An alternative to protected regions is the use of the *generation gap* pattern [Vli98, pp. 85ff]. Based on this pattern, manually written code can be added non-invasively by means of inheritance: The "hand-made" classes simply extend the generated classes. On regeneration, the code generator can safely overwrite the superclasses, and the manually written subclasses are not affected at all. Hence the code generator is less complex than for the protected regions approach, because it only has to ensure that, e.g., suitable visibilities in the generated code support the inheritance.

Besides partial RTE, there is also *full round-trip engineering* [Fra02, pp. 235f] which allows arbitrary changes to model and code along with a bidirectional synchronization of both. However, in practice, full RTE is very hard to realize due to the fact that "transformations in general are partial and not injective" [HLR08]. As a consequence, full RTE often only works if model and code are at the same level of abstraction [Sta+07, p. 45; KT08, pp. 5f]. This contradicts the very purpose of a model, that is, to be an abstraction of the code (cf. Sect. 2.2).

MDA is a prominent example of an approach that is frequently realized on the basis of RTE. Many code generators for UML, such as AndroMDA which is presented in more detail in Sect. 8.1, mainly produce stubs and skeletons that have to be completed manually. Hence lots of UML modeling tools like Together or Altova UModel provide support for RTE. As many UML models are very close to the code in terms of abstraction, even full RTE is possible – however, as already pointed out in Sect. 2.3.3, UML is often criticized exactly for this lack of abstraction.

Full Code Generation:

An alternative approach that aims at avoiding the problems arising from stub/skeleton generation and RTE is *full code generation* [KT08, p. 49 f]. This refers to the generation of fully functional code which does not require any manual completion. More precisely, the manual modification of the generated code is explicitly forbidden: Any change to the system has to be performed at the modeling level, followed by a regeneration of the code. As the code is never edited, the code generator can overwrite it blindly (similar to the superclasses of the generation gap pattern, see above) which strongly simplifies the generation. In this scenario, the generated code is considered a by-product, analogous to the results of a compiler for a programming language [Sel03].

Please note that full code generation usually is not equivalent to generating a full application, though in some cases the generated source code may already resemble a complete application or system. Typically, the generated parts coexist with other code and software components, such as hand-written code (e.g., specialized GUIs, legacy code, a domain framework in the sense of DSM), frameworks (e.g., a web framework like Struts [Apa11d]), libraries (e.g., a template engine like StringTemplate, see Sect. 2.4.2), or an application server like JBoss [Red11a].

It largely depends on the source language whether full code generation is possible or not. The challenge is to design the language in such a way that it contains enough information for the generation of complete code, but at the same time is not forced to align its abstraction level with the code.

For instance, the latter can be observed with Executable UML [MB02; Rai+04], which aims at making UML models executable via precisely defined action semantics, using a compliant action language like the Action Specification Language (ASL) [Ken03]. Although this technique improves the results of code generation, it comes at the cost of less abstract and more technical models: Executable UML is virtually using UML itself as a programming language. [KT08, pp. 56f]. Similar arguments apply to other approaches that, e.g., try to generate the dynamic aspects from collaboration diagrams [Eng+99].

One approach for achieving full code generation is specifically tailoring the language and the code generator to each domain, as, e.g., advocated by DSM and MDSD. This book will show that another solution is the combination of model-driven development and service-orientation that is proposed by the XMDD paradigm (cf. Chap. 3).

2.5 Quality Assurance of Code Generators

Just like any other software product, code generators have to be the subject of quality assurance measures such as verification and validation (V&V). Bugs in code generators may lead to drastic problems such as uncompilable code or unexpected behavior of the generated system. This is particularly unacceptable for safety-critical systems that can be found, e.g., in the automotive or aviation industry. In consequence, it is essential that the automated translation provided by a code generator is dependable and always leads to the desired results.

In compiler construction, there has been lots of research on V&V, including compiler verification (e.g., based on techniques like theorem proving [Str02; Ler06], refinement algebras [MO97], translation validation [PSS98; Nec00], program checking [GZ99] and proof-carrying code [Nec97]) as well as compiler testing [KP05]. In particular, the "verifying compiler", i.e., one that proves the correctness of the compilation result, has been the subject of a grand challenge proposed by Tony Hoare in 2003 [Hoa03]. Moreover, compiler verification in general is still an active topic (see, e.g., the workshop on "Compiler Optimization Meets Compiler Verification", COCV; or the conference on "Verified Software: Theories, Tools and Experiments", VSTTE).

Sect. 2.1 already pointed out that existing tools from the realm of compiler construction (e.g., parser generators) can be reused for the construction of code generators in MD* approaches. Similarly, insights and techniques from compiler verification often serve as the basis of V&V for such code generators. For instance, theorem proving is used by Blech et al. [BGL05] to verify the translation of statecharts to a subset of Java, and in the Gene-Auto [Rug+08] project for verifying the generation of C code from data-flow and state models. Ryabtsev and Strichman [RS09] apply translation validation to a commercial code generator that translates Simulink [The11] models to optimized C code. Denney and Fischer [DF06] propose an evidence-based approach to the certification of generated code that is similar to the ideas of proof-carrying code.

Concerning testing, Stürmer et al. described "a general and tool-independent test architecture for code generators" [Stü+07; SC04]. Sect. 6.3 further elaborates on this testing approach, as parts of it have been realized in the context of the Genesys framework presented in this book. Beyond the publications of Stürmer et al., the author could not find any further substantial research on code generator testing.

Stürmer et al. categorize V&V of code generators as *analytical procedures* [SWC05]. Apart from this, they also identify further approaches to the quality assurance of code generators termed *constructive procedures.* Such approaches advocate the implementation of code generators along the lines of systematic development processes. According to Stürmer et al., this includes, e.g., the adoption of standards like SPICE (Software Process Improvement and Capability Determination, ISO/IEC 15504).

2.6 Classification of Genesys

This section locates Genesys on the scale of approaches and techniques presented in the previous sections. For this purpose, it focuses on highlighting the differences and similarities – for any details on the single aspects of Genesys there will be cross-references to the corresponding chapters in this book.

As pointed out in Chap. 1, the Genesys approach propagates the construction of code generators on the basis of graphical models and services. This approach is, to the knowledge of the author, unique in the realm of code generation.

Generally, the advantages of service orientation are typically not exploited for building code generators. For instance, this is also true for the field of BPM, which is traditionally closely connected to the ideas of service orientation. Furthermore, it frequently features the combined use of graphical models and services (e.g., in BPMN, cf. Sect. 2.3.6). However, those notations are typically used for higher-level business processes, and not for lower-level technical domains such as code generation.

If a program written in a DSL is considered a model (cf. Sect. 2.2), one could argue that some approaches (e.g., MERL in MetaEdit+) indeed employ modeling for realizing code generators. However, none of the code generation approaches known to the author of this book uses *graphical* models for this purpose: Textual specifications of code generators are the rule.

A reason for this might be that code generation generally seems to be attributed to a lower level of abstraction. Code generators are mainly implemented by developers who are used to textual languages and APIs – so why bother them with graphical models and services? This book argues that the use of both can be highly beneficial for the development of code generators.

The previous sections showed that existing approaches are usually restricted to the use of specific code generation techniques (e.g., templates engines in AndroMDA, rule-based transformations in Spoofax, or the language Xtend in Xtext). In contrast to this, Genesys does not dictate which techniques or tools are used for building a code generator. This is a direct consequence of service orientation: Any tool or framework can be incorporated as a service and directly used in Genesys. Modeling on the basis of the available services is not fixed to any specific procedure, and thus the generator developer is free to choose any technique and modus operandi for the code generator.

For instance, most of the Genesys code generators exemplified in this monograph (cf. Sect. 4.2 and Chap. 5) employ template engines and thus can be considered template-based. Each template engine is an available service, so that the generator developer can freely select which engine should be used. He could even mix several template engines in one single code generator.

It should be noted that in order to obtain a clean separation of generation logic and output description (*Requirement S4 - Clean Code Generator Specification*), many Genesys code generators employ template engines in a different manner than typical template-based generators. For instance, as a convention in Genesys, advanced features of template languages such as control flow statements or function calls should be avoided: Instead the corresponding logic is specified explicitly in the code generator models, so that it can, e.g., be captured by verification tools (see Sect. 4.2.5). As a result of this convention, those Genesys code generators typically use rather small templates that are distributed over the code generator, producing code fragments that need to be assembled at some point of the code generation process. This is similar to, e.g., the rule-based transformation approach described by Hemel et al. [Hem+10], which employs a similar fragmentation of the output description.

Apart from separating generation logic and output description, a further advantage arising from this different use of template engines in Genesys is the fact that code generators can be source-driven and template-based at the same time. As mentioned above in Sect. 2.4.2, code generators employing template engines are typically restricted to target-driven transformation. However, because Genesys imposes no restrictions on the order in which the code fragments have to be produced, the generation of the output can be performed in a source-driven as well as in a target-driven manner, or even with a combination of both. This flexibility also helps to overcome the typical problems of template-based code generators that occur when dealing with multiple files (cf. Sect. 2.4.2). The Documentation Generator described in

Sect. 4.2 is an example which employs templates, is both source-driven and target-driven, and deals with multiple output files.

This book also shows examples of Genesys code generators which are not template-based at all. For instance, the *FormulaBuilder* (cf. Sect. 6.2.1) employs a rule-based transformation with a concrete form target, and the *BPEL Generator* (cf. Sect. 5.4.5) performs a transformation to an abstract form target and then serializes this to code.

Furthermore, this book illustrates the flexibility arising from service orientation by integrating and using the code generation framework AndroMDA as a service (in this case even paving the way for full code generation, cf. Chap. 8). Consequently, Genesys may be considered "a code generator construction kit which allows the (re)use and combination of existing heterogeneous tools, frameworks and approaches independent of their complexity" [JS11]. In this role, Genesys does not complement, but supplement and unify existing approaches.

Additionally, Genesys is not limited to any particular source language (see Chap. 7) or representation of the source language, like the bulk of language workbenches which strongly focus on textual source languages. Likewise, there are no restrictions of supported target languages whatsoever.

The development of code generators in Genesys is characterized by the reuse of existing components, as it relies on a library of models and services (cf. Chap. 4). Accordingly, Genesys strives for a balanced approach that aims at:

1. providing fast creation of code generators via customization and reuse, in contrast to, e.g., DSM and language workbenches, which usually achieve their high domain-specificity by developing an entirely new code generator for each domain (thus repeatedly starting from scratch), and at the same time
2. being more flexibly adaptable to different domains than, e.g., CASE or UML tools with their rather fixed and inextensible code generators.

As another major difference in comparison to other approaches, Genesys provides a holistic view on code generator construction that supports all phases including the specification, execution, generation, debugging, verification and testing of a code generator[1]. While specification, execution and generation are typically supported, facilities for debugging a code generator are more rare. Among the examples listed in the previous sections, only MetaEdit+ supports this by means of a dedicated tool [TK09], and in the case of code generators implemented with a programming language, existing debuggers can be used. However, for testing and in particular for verification, most approaches do not provide integrated and dedicated solutions.

Furthermore, Genesys aims at retaining simplicity along all phases of code generator development. Following *Requirement G3 - Simplicity*, the goal is that constructing a code generator demands learning as few languages as

[1] For specification, execution, generation and debugging see Chap. 4 and 5, for verification and testing see Chap. 6

possible. In other approaches, the knowledge of multiple languages (or at least dialects of a language) are required, apart from the actual source and target language of the code generator. For instance, the language workbench Xtext has separate languages for specifying grammars, transformations and workflows of transformations [Ecl11h]. Genesys uses the same simple modeling language (cf. Sect. 3.2.2) for all artifacts required in the single phases. In consequence, artifacts like test cases, test suites (cf. Sect. 6.3) or constraints (cf. Sect. 6.2) are specified by means of the same language employed for developing the actual code generators. Aside from this, only a (freely selectable) template language might have to be learned, given the case that a template-based code generator is to be developed.

Concerning verification, Genesys is also unique in that it applies model checking for proving the correctness of code generators relative to a set of constraints. Although in particular model checking and another facility called *local checking* (i.e., checking of constraints attached locally to single services, cf. Sect. 6.1) are in the focus of this book, other verification techniques can be easily incorporated into Genesys (cf. Sect. 10).

The reference implementation of the Genesys approach presented in this monograph is conceptually and technically based on another MD* approach called XMDD and its tool incarnation jABC (cf. Chap. 3). Sect. 3.5 evaluates the feasibility of other MD* approaches and tools with regard to their aptitude for realizing the requirements of the Genesys approach (cf. Sect. 1.1), and in doing so it illustrates why XMDD and jABC are a suitable basis for reaching those goals.

Finally, the combination of XMDD, jABC and Genesys can be considered a realization of GP (cf. Sect. 2.3.2). In this combination, services resemble the elementary implementation components situated in GP's solution space, and models provide a particular configuration of services in the problem space. Those models are a suitable basis for the evolution of system families, as exemplified with a family of code generators in Chap. 5. Variability can be specified by means of the variant management features presented in this book (cf. Sect. 4.1.4 and 10). The code generators provided by Genesys in conjunction with a library of constraints (cf. Sect. 3.1) embody GP's configuration knowledge.