
List of Tables

3.1	Complex built-in data types in jABC	49
4.1	Services for data type mapping	82
4.2	Services for identifier generation.....	84
4.3	Service for the generation of variants	87
5.1	Java mapping for complex jABC data types	114
5.2	Quantitative comparison of the different Java code generators	128
5.3	Experimental performance results for classes produced by the different Java code generators.....	134
5.4	Size of the generated Java source classes	135
5.5	JME mapping for complex jABC data types	145

Motivation and Fundamentals

Introduction

Today's software systems and the platforms they reside on "are often among the most complex engineering systems" [Sel03]. In their day-to-day work, software engineers are faced with a huge and constantly changing variety of languages, tools, frameworks and platforms that need to be mastered.

Abstraction is a traditional and very powerful means for shielding software engineers from this complexity. For instance, operating systems hide the complexity of underlying hardware platforms, and compilers allow the use of high-level programming languages that abstract from low-level languages such as assembly or machine code.

At an additional level of abstraction, *models* allow the specification of a software system independent of the concrete technologies that are used for its actual implementation. However, in order to fully exploit the potential of models, it is not sufficient to use them for documentation and visualization purposes only (called *model-based* development by Stahl et. al [Sta+07, p. 3]). Instead the gap that arises between the model of a system (the abstraction) and a corresponding implementation (a concrete incarnation of the abstraction) has to be bridged.

This is the task of *code generators*, which automatically derive an implementation from the model, and which thus relate to models in the same way in which compilers relate to high-level languages. Consequently, models are promoted to primary development artifacts. Ideally, the availability of automatic code generation relieves the developer from ever having to deal with the resulting code, which is considered a by-product just like the results produced by compilers. Due to the prominent role of models, corresponding approaches to software development are called *model-driven*. Today this realm has many incarnations referred to by a plethora of acronyms such as MDD, MDE, MDSD, MDA, DSM (cf. Chap. 2 for details on the single abbreviations) and so forth. Völter [Völ09] coined *MD** as an umbrella term that conveniently comprises all approaches to model-driven development.

Apart from bridging the gap between models and implementations, code generation provides further advantages. Herrington [Her03, p. 15], e.g., refers

to the high quality of generated code, which is usually more consistent than hand-written code. For instance, a code generator automatically and consistently propagates bug fixes and other modifications to all corresponding parts of the code. Furthermore, Herrington points out that the generation of cumbersome boilerplate code allows the software engineer to concentrate on the important design aspects.

With the evolution of MD* approaches, code generation has successively gained in importance. Whereas in earlier approaches, such as Computer-Aided Software Engineering (CASE) [CNW89], code generators usually were fixed and static parts of general-purpose development environments, today's approaches increasingly focus on domain-specificity: From the modeling language through to the required tooling, all aspects of a development environment are tailored to each domain, which usually in particular includes the construction of dedicated code generators.

Consequently, there is a high demand for approaches that enable a simple and fast development of code generators. Currently, lots of techniques are available that support the specification and implementation of code generators, such as dedicated Application Programming Interfaces (APIs), template engines or rule-based transformation engines (cf. Sect. 2.4). All those techniques have in common that code generators are either directly programmed or described by means of textual specifications.

This book presents a novel approach called *Genesys*, which suggests the *graphical* development of code generators on the basis of *models* and *services*. It will be shown that this unique approach provides significant advantages for the construction and evolution of code generators.

Why models?

Just like for any software system, the advantages of models can also be applied to code generators. As code generators tend to be very complex, they can in the same way benefit from the abstraction provided by models. Especially *hierarchical* models (i.e., supporting a mechanism that allows embedding models into each other) are a powerful means for mastering complexity. Furthermore, apart from its structuring character, hierarchy enables the separation of concerns, e.g., by only showing those parts of the model hierarchy which are of current interest to the developer.

Another benefit arising from the abstraction provided by models is portability: A code generator can be developed independently from a particular implementation language or host machine. Via suitable generator generators, the same code generator can be translated for and deployed to arbitrary host systems without the need of adapting the generator's design.

Finally, models are amenable to formal verification methods such as model checking (cf. Sect 1.1), which allows the application of powerful tools that increase the reliability and robustness of code generators.

Why graphical models?

According to Sendall, “there are perceived cognitive gains” [SK03] when using graphical notations, and available research on visual programming languages underpins this (see e.g., [Bla96; Whi97; Bla01; Moo09]). For instance, a frequently used argument (e.g., [Kle08, p. 5; Moo09]) is that textual notations are one-dimensional due to their purely sequential character, whereas the spatial arrangements used in graphical notations may be more complex, but at the same time lead to a greater expressiveness. Furthermore, Moody [Moo09] points out that graphical notations are usually processed in parallel by the human mind in contrast to the serial processing of textual notations. Finally, Schmidt [Sch06] argues that graphical representations help flatten learning curves.

Why services?

Services [MSR05] are reusable units with simple and unified interfaces. As their usage does not require any knowledge about their actual implementation, services are, just like models, a suitable means for hiding complexity. Furthermore, assembling a code generator from services adds to its modularity and adaptability: Single services can be easily replaced by newer or alternative versions, thus allowing *agile* development and evolution. Existing code generation techniques, tools and frameworks can be made available as services, which can be freely used and combined in order to realize complex code generators. Moreover, models that are assembled from services are typically *executable*. Apart from supporting rapid prototyping and easy debugging, constructing code generators as executable models also enables the application of bootstrapping (cf. Sect. 1.1).

Genesys is a *general* approach for modeling code generators for arbitrary source and target languages. In particular, it is designed to support incremental language development on arbitrary metalevels (cf. Chap. 7). This monograph shows the feasibility of the Genesys approach by means of a fully-fledged reference implementation, which has been field-tested in a large number of case studies. The conceptual and technical basis of this implementation is given by the *Extreme Model-Driven Development* (XMDD) paradigm and its tool incarnation *jABC* (cf. Sect. 3), which enable model-driven and service-oriented development according to the mindset described above. The reference implementation meets a number of requirements which have been identified for the realization of the Genesys approach.

1.1 Requirements of the Genesys Approach

As described above, model-driven development and service orientation are the basic principles of the Genesys approach. Based on these two key demands, several general requirements can be formulated:

Requirement G1 - Platform Independence

The abstract nature of models allows designing a code generator independent of a specific programming language or particular host system on which it will be executed. Thus the same code generator model may be translated for execution on different host systems. Using services as building blocks of models further strengthens this abstraction: For the generator developer, those services are black boxes, and their usage does not require any knowledge about their concrete implementation. In order to guarantee a code generator's translatability for multiple platforms, service implementations need to be easily interchangeable, but by all means transparent to the generator developer. Furthermore, the employed service mechanism must not pose any restrictions on what can be made available as a service, so that there are no limitations on which libraries or tools (e.g., template engines, cf. Sect. 2.4.2) can be used for composing a code generator. Finally, a code generation framework must not be restricted to code generators that only support particular source or target languages.

Requirement G2 - Reusability and Adaptability

As programming languages, platforms and libraries change rapidly and new ones emerge frequently, code generator frameworks need to support *fast creation* and *adaptation*. Facilitating this kind of agility is one of the key concerns of service orientation: Services are made available in repositories in a way that allows to reuse them among different application contexts, thus avoiding the proverbial "reinvention of the wheel". This reusability, which is self-evident at the service level, is also desirable for entire models. If models themselves are easily reusable, code generators and their features can, once modeled, be collected in a repository just like services, so that modelers benefit from work that has already been done before. Furthermore, as every new code generator in turn contributes reusable assets to this repository, the potential of reuse is growing continuously.

Requirement G3 - Simplicity

In recent code generation approaches, often a huge variety of different languages is involved in the development of a code generator (cf. Sect. 2.4), such as template languages, transformation languages, grammar notations or constraint languages. This demands a high learning effort prior to getting started and inevitably slows down development. Thus it is desirable to keep the number of required formalisms, modeling notations and languages as small as possible in order to reduce the complexity of using the framework, of course without limiting its overall functionality. Choosing services as basic building blocks for models already is a first step towards simplicity, as services should have standardized and very simple interfaces. When replacing one service with another, it is not necessary to learn a new API or even programming language, as it is often the case when substituting libraries at the code level. Furthermore, as

already mentioned above, services can be used without knowing any details about their actual implementation or complexity.

Requirement G4 - Separation of Concerns

As code generators may get very complex, a suitable modeling language and its corresponding tools should support the separation of concerns. A generator developer should be able to focus on particular aspects rather than being confronted with the full amount of information all along. *Hierarchical models* have already been mentioned above as a possible way of supporting the separation of concerns.

Requirement G5 - Verification and Validation

Software verification and validation (often referred to as “V&V”) are key activities for ensuring that a system meets all previously specified requirements and needs, and that it is built in an appropriate and correct way. When writing source code in an Integrated Development Environment (IDE), usually several checking mechanisms are executed in the background, performing, e.g., syntax checks or static code analysis. If any problems are detected, the developer is immediately alerted. Similar checks are often also supported by today’s modeling environments. Given a metamodel that describes a modeling language’s abstract syntax and static semantics (cf. Sect. 2.2), a corresponding editor with suitable checking facilities can be easily provided. Apart from syntactic checking, formal methods like model checking (cf. Sect. 3.4) go further by verifying whether a model conforms to a set of given constraints, often tailored to a specific domain. Provided that a suitable modeling language is chosen (i.e., one that is supported by existing model checkers) and a library of constraints for code generation is created, model checkers can perform sophisticated verification and thus increase the robustness and reliability of code generators. This verification potential goes beyond anything supported by most code generation approaches today (cf. Sect. 2.5). Beyond such checking mechanisms, *testing* is another important facet of V&V that needs to be supported by a code generation framework.

On the basis of those fairly general demands, several more specific requirements arise, especially when considering the current state of the art in code generation (cf. Chap. 2):

Requirement S1 - Domain-Specificity

Being themselves an enabling technique for domain-specific modeling as outlined above, it is also desirable to construct code generators in a way that respects the domain knowledge of the generator developer. Instead of using a general-purpose modeling formalism, the selected modeling language should be adaptable to a given domain, including suitable terminology and visualization. This adaptation could be initially performed by a domain expert, or even be automated on the basis of a given metamodel that describes the domain. A customized modeling language tailored to

the domain knowledge of the modeler greatly improves the simplicity of the approach (cf. *Requirement G3 - Simplicity*).

Requirement S2 - Full Code Generation

A common technique that is involved in several code generation approaches is *round-trip engineering*. Essentially, this term refers to the automatic tool support for keeping models in sync with code generated from them. This is especially necessary when generated artifacts have to be manually modified or completed by developers. Changes in the generated artifacts need to be propagated to the corresponding models and vice versa, while assuring at the same time that no manual work is lost or overwritten. Implementing automatic tool support for this task is very cumbersome and increases the complexity of involved code generators (Sect. 2.4.4 elaborates on that). Consequently, it is desirable to avoid the need of round-trip engineering and to employ approaches that enable *full code generation*. Accordingly, all modifications are exclusively performed on the models, while generated artifacts are considered a by-product that never has to be touched. If there are any changes to the models, then the corresponding artifacts are simply regenerated. This deliberate avoidance of a bidirectional synchronization of models and generated artifacts considerably eases the work of a corresponding code generator and the overall development process (cf. Sect. 2.4.4).

Requirement S3 - Variant Management and Product Lines

The evolution of code generators based on reuse and adaptation (cf. *Requirement G2 - Reusability and Adaptability*) needs to be facilitated by corresponding tools. This includes tool support for the definition and creation of variants, using existing code generators and features as patterns for new *product lines*.

Requirement S4 - Clean Code Generator Specification

A typical code generator usually consists of two main aspects. First, its *generation logic* realizes, e.g., the traversal of the input model's elements and collects the data required for code generation. Second, the code generator includes an *output description* that specifies the resulting code or markup (cf. Sect. 2.4). Especially modern template languages often mislead to mixing up those two aspects, which may make it harder to understand, maintain and adapt a code generator, and to use it as a pattern for other code generators (cf. *Requirement G2 - Reusability and Adaptability* and *Requirement S3 - Variant Management and Product Lines*). Thus it is desirable to establish a way for clearly separating generation logic and output description when designing a code generator (cf. *Requirement G2 - Reusability and Adaptability*).

Requirement S5 - Bootstrapping

Bootstrapping (see Sect. 2.1) is a common development technique that emerged from compiler construction. Basically, a new compiler is developed in several stages by incrementally adding target language features and applying existing compilers to each other. As this approach is

well-established and proven, it should be supported by a modern code generation framework.

Requirement S6 - Tool-Chain Integration

As code generators are usually built to be integrated into tools or productive development setups consisting of certain tool-chains, a code generation framework and the code generators based on it need to be compatible with build management tools like Apache Maven [Apa11b].

1.2 Organization of the Book

This book is divided into three main parts:

Part I:

After this introductory chapter, Chap. 2 presents the state of the art in code generation. Besides establishing terminology that is required for the rest of the monograph, the overview of the fundamentals of code generation provided by this chapter is at the same time intended as a presentation of related work. Accordingly, the chapter finishes with a comparison of Genesys with other approaches. Afterwards, Chap. 3 introduces the XMDD paradigm as well as jABC along with several plugins that play a central role for this book. The chapter also compares other MD* approaches with XMDD/jABC and evaluates their suitability as a basis of the Genesys approach.

Part II:

This part elaborates on the reference implementation of the Genesys approach called the *Genesys framework*, and on the case studies that were performed in order to field-test it. Chap. 4 presents the Genesys framework along with its services and tooling. Furthermore, it exemplifies the use of the framework by describing the construction of a simple code generator. Chap. 5 elaborates on a collection of case studies that aimed at providing various code generators for jABC itself. Chap. 6 illustrates the verification and validation of code generators in the Genesys framework by means of examples from the jABC case studies. Another case study is presented in Chap. 7, which describes the use of Genesys for the construction of domain-specific code generators for Eclipse Modeling Framework (EMF) on the basis of services that are generated from a given metamodel. Finally, Chap. 8 discusses the last case study that deals with the integration of services into Genesys, exemplified by means of the code generation framework AndroMDA.

Part III:

Chap. 9 sums up and draws several conclusions. In particular, it revisits the requirements of the Genesys approach and shows how they have been met by the reference implementation. In the end, Chap. 10 elaborates on future plans and possible extensions of Genesys.

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.

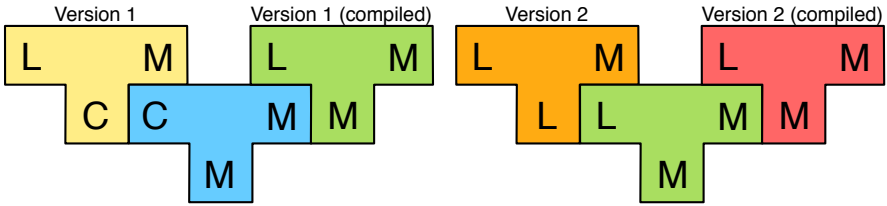


Fig. 2.1. Simple Bootstrapping Example: Getting a Native Compiler for Language L

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.

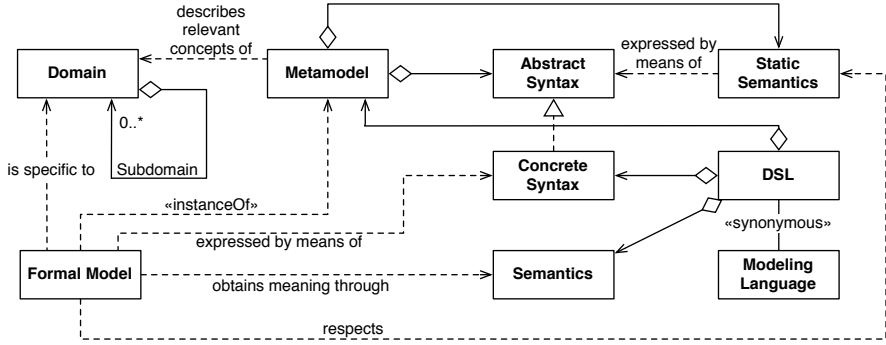


Fig. 2.2. Basic MD* terminology (by Stahl et al. [Sta+07, p. 28], translated into English)

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]).

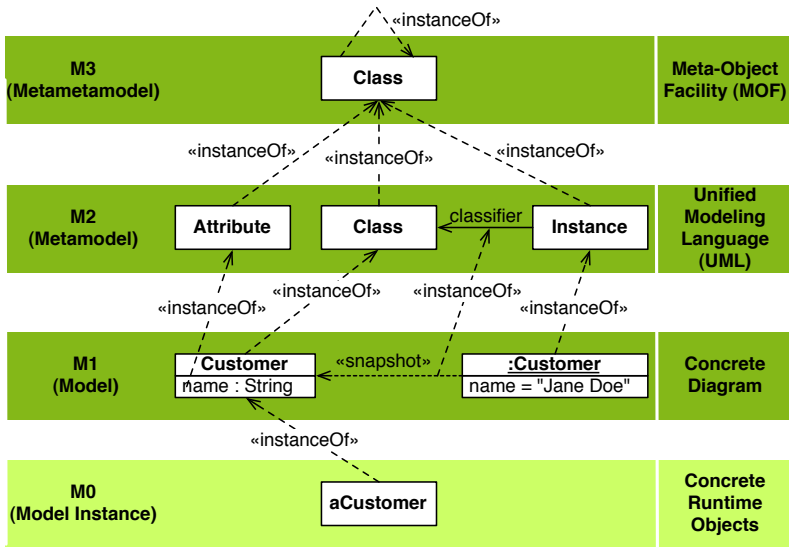


Fig. 2.3. Four-Level Example of MDA’s Metamodel Hierarchy (based on [Obj10a, p. 19])

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 metamodel (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], Spoofox [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 Helsén [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.

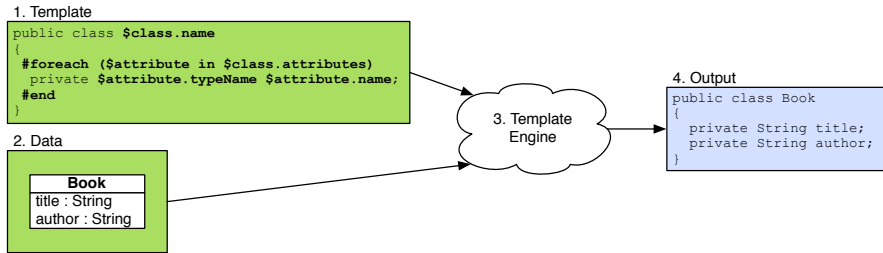


Fig. 2.4. Using a template engine for code generation

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 Spoofox) 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 uncompileable 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 GeneAuto [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 Spoofox, 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 generator¹. 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 [Ec11h]. 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.

Extreme Model-Driven Development and jABC

This chapter introduces the jABC framework and the underlying *Extreme Model-Driven Development* (XMDD) paradigm, which form the technical and conceptual (respectively) basis for the reference implementation of the Genesys approach described in this book. Historically, the demand for code generation in jABC provided the initial motivation for starting the Genesys project. In the beginning, the project started off as one of many application scenarios of jABC and XMDD. However, during its evolution, the approach showed great potential for application beyond the jABC context, so that the scope of the Genesys project has broadened.

Nevertheless, XMDD and jABC remained the integral core of the Genesys framework. Accordingly, reading the following sections is highly recommended, as they introduce the notions and concepts that are required for understanding all remaining chapters of this book. Sect. 3.1 first introduces the XMDD approach, which establishes a basic mindset for model-driven and service-oriented software engineering. Afterwards, Sect. 3.2 introduces jABC as a concrete framework and toolset for software development along the lines of XMDD. Sect. 3.3 describes the execution of models in jABC, and Sect. 3.4 elaborates on jABC's support for system verification via model checking, which has been used intensively in the Genesys framework (cf. Chap. 6). Finally, Sect. 3.5 discusses the feasibility of jABC/XMDD as a basis for realizing the Genesys approach in comparison to related approaches.

3.1 Extreme Model-Driven Development

Today's software projects are facing several kinds of gaps that may lead to serious problems or even to failure. First, there are *social gaps* between people involved in developing a software product. Those gaps may be of cultural nature, e.g., for development teams scattered globally, or they result from miscommunication between project members with differing mindsets and professional backgrounds, such as customers, developers and the management.

Charette [Cha05] lists “poor communication among customers, developers, and users” as one of the most common reasons for failed projects. Cerpa and Verner add further communication-related factors such as “customer/users [sic] had unrealistic expectations” or “process did not have reviews at the end of each phase” [CV09].

Second, as described by Margaria and Steffen [MS08], *system gaps* hamper the construction of software, especially of heterogeneous large-scale systems that cross organizational boundaries. Typically, such systems are composed of components, such as libraries or entire products like enterprise resource planning (ERP) systems, that evolve independently of each other, e.g., because they are created by different manufacturers. Consequently, updates of single components are very problematic, as they may affect the correct functionality of the overall system in an unexpected – and often also unpredictable – way, thus making the problem diagnosis a very cumbersome task.

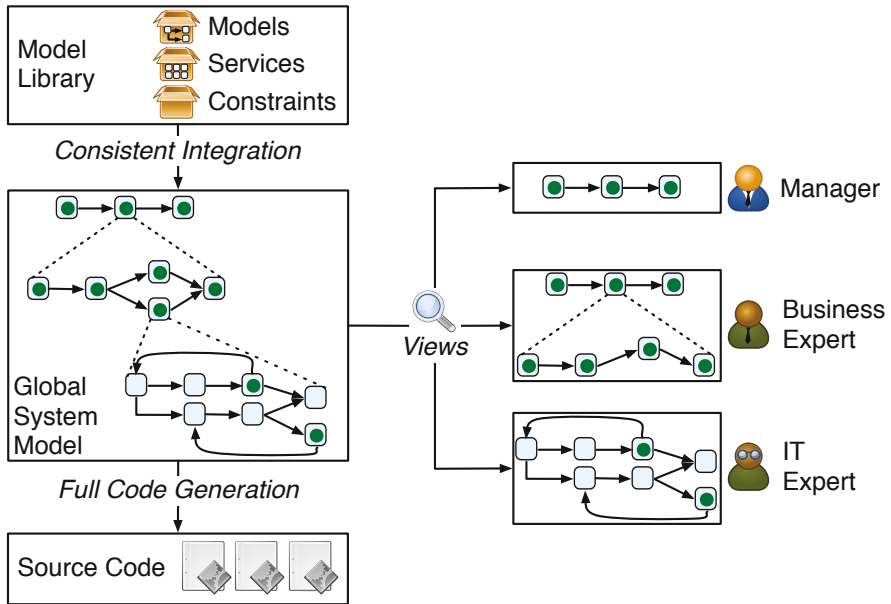


Fig. 3.1. The Extreme Model-Driven Development (XMDD) approach

The XMDD [MS08; MS09a; MS09b] approach illustrated in Fig. 3.1 has been proposed by Steffen et al. for bridging those gaps. In order to achieve this, it combines the ideas of model-driven development, service orientation, extreme programming and aspect orientation [MS09b]. By promoting models to central and primary development artifacts, XMDD exploits the fact that models typically raise the level of abstraction (cf. Chap. 1). Moreover, the

approach postulates that the integration of a system is performed at the modeling level rather than at the level of software components [MS08].

As shown in Fig. 3.1, modeling is performed on the basis of a *library* of models, which are combined to one large artifact, the *global system model*, that reflects the actual system. Due to the approach's focus on this single central artifact it is also called the *One-Thing-Approach* (OTA) [SN07; MS09a].

Integrating the system at the modeling level provides the significant advantage that models are amenable to formal methods. Accordingly, the consistency of the global system model (i.e. the valid combination of its constituent models) can be ensured by means of corresponding tools that perform, e.g., model checking (see Sect. 3.4). Such mechanism help overcoming the compatibility and interoperability issues of software components outlined above.

A necessary condition for this *modus operandi* is the consequent restriction of any system changes to the modeling level. As visible in Fig. 3.1, the global system model is automatically translated into code for a desired target platform via full code generation (cf. Sect. 1.1). Due to the fact that this generated code is complete (i.e. no stubs or skeletons), it can be considered a by-product that does not have to be modified. Any system changes are directly performed at the modeling level by adapting the global system model, followed by a regeneration of the corresponding code.

As the code generator does not need to take care of any manual changes on the code, results from old generation runs can be overwritten safely. Accordingly, there is no need for any round-trip engineering mechanisms which may introduce additional complexity or problems that jeopardize the system's consistency (cf. Sect. 2.4.4). XMDD shares this tenet with other MD* approaches such as MDSD and DSM (cf. Sect. 2.3).

The global system model is used as a common basis for *collaboratively* designing the actual system. As a central objective of XMDD, this process is not solely performed by developers. Instead the customers and/or business experts are continuously involved in the entire project evolution, with the global system model being the shared language for bridging the social gaps described above. Again, the abstract nature of models is beneficial for achieving this, as any technical details of the final target platform are faded out. However, several additional mechanisms are required in order to provide a model that is capable of combining both the developer's technical perspective and the business expert's functional perspective of a system.

One of such mechanisms in XMDD is *hierarchical modeling* [Ste+97]. As visible in Fig. 3.1, the global system model is constructed in a hierarchical fashion, which, apart from reducing the model's complexity through modularization, enables a collaborative creation of the model by means of iterative refinement. Starting at a rather abstract level, which only specifies the global tasks or features of the system, the model is continuously refined and enriched, getting more and more concrete with each hierarchy level. Finally, the models at the lowest level consist of atomic *services*, which are

elementary building blocks of models, and which represent real functionality of the system.

This functionality is either reused from existing systems or software components (e.g., commercial off-the-shelf or libraries), or it is implemented by a developer. In order to be able to integrate the represented functionality in different application domains, services are usually configurable. However, at the modeling level, the details of the service implementations are not visible, so that the “system development becomes in essence a user-centric orchestration of intuitive service functionality” [MS08]. This orchestration of “real” functionality has another important advantage: Models composed of services are immediately *executable*. Among other things, this enables rapid prototyping, debugging and tangibly trying out the modeled system, even at very early stages of development. As shown at the top left of Fig. 3.1, services are an integral part of the model library.

Analogous to the reusability of atomic services, hierarchical modeling also allows the reuse of entire models. Once created, a model can be seen as a ready-made *aspect* or *feature*, which becomes a part of the model library and thus can be reused in other application scenarios. Consequently, the model library plays the role of a steadily growing repertoire: With each new application domain or developed system, this library is enriched by new models and services, thus constantly increasing the potential of reuse for any following projects.

This steady growth even applies to *constraints*, which are, besides models and services, the third part of XMDD’s model library. Constraints specify the consistency rules of the global system model and its constituent parts. Typically, there are two types of such constraints: local and global constraints. *Local constraints* concern the atomic parts of a model, i.e. the contained services. In contrast to this, *global constraints* relate to an entire model and usually ensure the consistent interplay of all contained models and services, in order to guarantee, e.g., well-formedness and executability.

Both types of constraints are verified by corresponding check tools such as model checkers (see Sect. 3.4) for global constraints. With each such constraint that is added to the library, the impact of the check tools on the integrity of the global system model as well as the automatic guidance of the user while modeling is increased, a process which is also referred to as *incremental formalization* [Ste+96; MS06].

Another mechanism for facilitating different perspectives on the global system model in XMDD is the support of *views* [SM99]. Such views are typically provided by means of transformations on the representation of the global system model. Fig. 3.1 shows some examples of different views:

- The manager’s view only contains the topmost hierarchy level of the model, as he is only interested in the global features or process of the system.
- The business expert’s perspective on the system is more detailed. He is able to see more hierarchy levels, but some of the models may be simplified

(e.g., by hiding error paths). Furthermore, his view does not include the lowest hierarchy levels that contain the concrete services.

- The IT expert's view contains the most concrete models at the lowest hierarchy levels.

MDA also captures different perspectives on a system with its concepts of CIM, PIM and PSM (cf. Sect. 2.3.3). However, in MDA these different models are connected by means of transformations of the *actual* model structure. The views in XMDD are *projections* of the global system model. This means that when a view is created, the global system model itself is not altered, but only its representation to the user.

By means of hierarchical modeling and views, the global system model becomes a suitable basis for the collaborative system development outlined above, as the perspectives of all involved persons are adequately supported. Apart from bridging the social gaps, the close cooperation with the customer/business experts in XMDD leads to shorter feedback cycles and a higher flexibility for dealing with changing requirements [MS09b]. XMDD shares those objectives with other agile approaches such as Extreme Programming¹ [BA04] or Scrum [SB01].

3.2 jABC

jABC [Ste+07; MS08] is a highly customizable Java-based framework that realizes the tenets of XMDD described above. Previous versions were based on C++, and the earliest precursors appeared almost two decades ago [Ste+94; SM99]. Currently, the framework is developed and maintained by the Chair of Programming Systems at the TU Dortmund.

jABC provides a tool that allows users to graphically develop systems in a behavior-oriented manner [MS06] by means of models called *Service Logic Graphs* (SLGs). As advocated by XMDD, SLGs are constructed hierarchically, and their elementary building blocks represent concrete services. Those building blocks are called *Service Independent Building Blocks* (SIBs). The jABC tool provides facilities for composing and manipulating SLGs as well as for instantiating and configuring the contained SIBs. Further functionality can be added by means of plugins.

Fig. 3.2 shows a screenshot of the tool's user interface, which consists of three main areas (corresponding to the numbers in the screenshot):

1. *Project and SIB browsers*: The project browser (not visible in the figure) provides an overview of all available projects, which are the basic organization units in jABC. Each system that is under development is reflected by a corresponding project which collects all models, services and constraints that are of interest for and employed by this system. For enabling the collaborative work on a project, its contents are usually

¹ In fact, the "X" in XMDD is a reference to Extreme Programming.

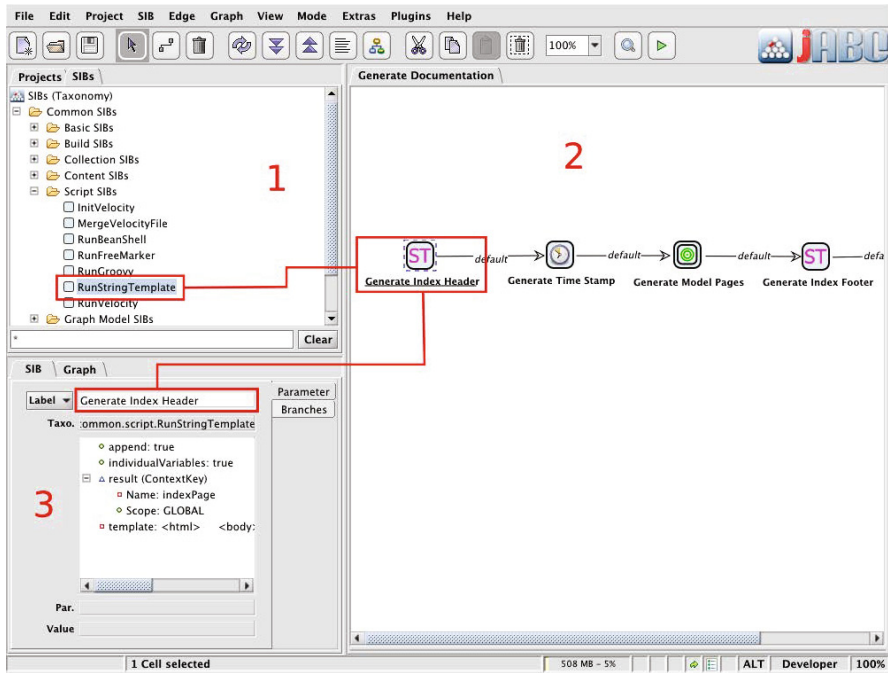


Fig. 3.2. The jABC user interface [JSM10]

managed by a revision control system such as Subversion [Apa11e]. Besides the project browser, the SIB browser enlists all services that can be used in the current project. The corresponding SIBs are organized by means of a *taxonomy*. In this context, taxonomies are graphs in which the sinks represent services and the intermediate nodes are groups or categories that subsume several services according to common properties or characteristics. Such a taxonomy is usually adapted to fit the current application domain in terms of which SIBs are visible (and thus applicable), and how they are named and organized. As visible from Fig. 3.2, jABC's SIB browser displays taxonomies as simple tree structures.

2. *Canvas*: On the canvas, the user graphically models the SLGs that represent the actual system. After selecting a service from the SIB browser (1), it can be instantiated by dragging it onto the canvas, where it is integrated into the SLG.
3. *Inspectors*: The inspectors provide detailed information on the currently displayed SLG and on its constituent parts. For instance, the *SIB inspector* allows to view and modify the current configuration of a particular SIB in the SLGs. This is visible in Fig. 3.2: The SIB inspector shows the configuration of the SIB **Generate Index Header** which has been selected in the canvas. As a further example, the *Graph inspector* allows the configuration of an entire model. Furthermore, it supports the

modification of metadata associated with an SLG, such as its name. Finally, plugins are able to add further inspectors that provide specific information and functionality.

Usually, when working with jABC in a team, several roles with different responsibilities can be identified [MS06;MS04]. First, the *application expert* or *business expert* has deep knowledge about the tasks and processes of a particular application, and he uses jABC for modeling this knowledge as SLGs. For this task, the application expert does not have to be familiar with the target platform, i.e., the technical infrastructure of the resulting system. In particular, no programming skills are required.

Second, the *domain expert* has detailed knowledge about the application *domain* including corresponding concepts and terminology. At the beginning of a new project, the domain expert customizes jABC in order to produce a variant that is tailored to the concrete domain and that optimally fits the needs of the application expert. For instance, this customization includes the adaption of the SIB taxonomy by selecting, naming and categorizing services on the basis of domain-specific terminology and characteristics. Further customization tasks are the selection of suitable plugins as well as the specification of additional domain knowledge such as supported data types or constraints [NLS11]. Finally, and most relevant in the context of this book, the domain expert selects required code generators for the translation of the SLGs for the desired target platform. If no appropriate code generator is available, he even may play the role of the *generator developer* (cf. Chap. 4) and use the Genesys framework in order to create a code generator that is tailored to the requirements of the chosen target platform. Accordingly, it may be advantageous (though not mandatory) if the domain expert is also roughly familiar with the technical characteristics of the target platform.

Third, the *IT expert* or *SIB expert* is technically versed and usually a classical software developer. Using his IDE of choice (e.g., Eclipse), he creates new SIBs on demand by integrating existing services or by implementing new ones. The SLGs created in collaboration with the application expert provide him with corresponding requirements. Accordingly, the IT expert also participates in the actual modeling, in particular in the creation of the more concrete models on the lower hierarchy levels (cf. Sect. 3.1). Furthermore, the IT expert is responsible for the software infrastructure and the runtime environment of the target platform.

The customization mentioned in the context of the domain expert is a central concept in jABC, as it provides the means for flexibly using the framework in arbitrary application scenarios. In contrast to the DSM approach described in Sect. 2.3.4, domain-specificity in jABC is not achieved by constantly creating entirely new languages, but by adapting the actual framework to the specific needs of the domain. The feasibility of this modus operandi is witnessed by a large variety of domains that have been covered in practical projects with jABC. For instance, jABC has been used for modeling telecommunication services [SM99], experiments in

bioinformatics [MKS08; LMS08], large-scale web applications [KM06], remote configuration management [BM06], supply chain management processes [Hör+08], robot control programs [Jör+07], game strategies [BJM09], test cases [MS04; Raf+08], compiler optimizations [MRS06], property specifications [JMS06] and web services [Kub+09]. This book presents code generation as another application domain for jABC.

The following sections elaborate on jABC’s core constituents that realize the ideas of XMDD: Sect. 3.2.1 further describes the concept of SIBs, Sect. 3.2.2 focuses on SLGs and Sect. 3.2.3 enlarges upon plugins. Parts of those sections are based on a previous publication [JSM10].

3.2.1 Service Independent Building Blocks

Service Independent Building Blocks (SIBs) are the elementary building blocks of models in jABC. The notation originates from the telecommunication realm [IIT97; IIT93]. It refers to a SIB being an abstract representation that is independent of

- a) its context of usage, i.e., reusable for composing different applications or systems (which in turn can be considered services from a compositional perspective [IIT97]) spanning arbitrary domains, and of
- b) the technical realization of the actual service functionality or behavior it stands for.

As already pointed out in Sect. 3.1, the granularity of the service represented by a SIB is arbitrary. It ranges from low-level functionality like string concatenation or database access to remotely available web services, or even to the interaction with more complex systems such as ERP software.

The Application Expert’s View on SIBs

From the perspective of an application expert who is modeling a system, the concrete manifestation of the represented services is entirely transparent and irrelevant: In order to use a SIB, it is only necessary to know which behavior it represents, but not how the behavior is implemented. As advocated by XMDD, application experts resort to a modeling repertoire that contains ready-made libraries of SIBs, which can be used as simple black boxes.

jABC already ships with a library of such SIBs, the *Common SIBs* [TU10], that represent very general and basic services that are of interest for almost any application domain. The Common SIBs are organized as bundles according to their tasks. For instance, the “IO SIBs” represent I/O functionality such as reading text from a file or creating directories, and the “Collection SIBs” provide services that deal with different collections such as hash tables or lists.

In order to enable an intuitive usage in jABC, SIBs have a simple interface [MS04] that provides:

- a set of *parameters* which enable the configuration of the SIB's behavior,
- a set of *branches*, which reflect the possible execution results of a SIB and which are used to connect SIB instances in a model via directed edges (cf. Sect. 3.2.2),
- an *icon* and a *label* which are used for visualizing the SIB in the canvas, and
- a *documentation* that informs the user about the behavior represented by the SIB and about the purpose of its parameters and branches.

In order to illustrate this, Fig. 3.2 highlights an example of a SIB used in a model displayed in the canvas (2). From the SIB browser (1) it is visible that the SIB is an instance of `RunStringTemplate`, which is categorized as part of the Common SIBs bundle called “Script SIBs”. `RunStringTemplate` integrates the template engine `StringTemplate` (cf. Sect. 2.4.2) as a service. Accordingly, the task of this service is the evaluation of a template. The corresponding SIB instance in the canvas is labeled **Generate Index Header**, and there is another instance of the SIB contained in the model (labeled **Generate Index Footer**), which illustrates the reusability of the building blocks. The icons and labels that are used to visualize the SIB instances in the canvas can be customized by the application expert.

By selecting a particular SIB instance in the canvas, its details are displayed by the SIB inspector (3). As visible in Fig. 3.2, the SIB `RunStringTemplate` provides four parameters, with one of them (“template”) being the template that should be evaluated by `StringTemplate`. Furthermore, the SIB has two branches (not visible in the figure): *default*, reflecting the case that the template evaluation succeeded, and *error*, indicating that the template could not be evaluated (e.g., due to syntax errors).

In order to enable SIB instances in a model to communicate with each other, i.e., to share data, the concrete service implementations keep track of an *execution context* which acts as a shared memory. Technically, this context is like a hash table containing a set of key-value pairs. Thus a SIB instance is able to read and manipulate data that has been stored in the context by other SIB instances, provided that both SIB instances agree on the key which identifies the data. While the concrete implementation of the execution context is irrelevant for (and invisible to) the application expert, it is his responsibility to specify the keys used by SIB instances for accessing shared data. These keys are specified by means of special SIB parameters. For instance, the SIB instance shown in Fig. 3.2 provides a parameter “result” (3) which specifies the key used to store the evaluation result of the `StringTemplate` service in the execution context (in the example, this key is `indexPage`). Sect. 3.3.2 elaborates on the concept of the execution context in more detail.

The IT Expert's View on SIBs

Among other tasks (cf. Sect. 3.2), the IT expert provides the application expert with required SIBs, either by continuously extending the ready-made

libraries such as the Common SIBs, or as a reaction to a direct request. Implementing a SIB basically consists of two parts: the SIB itself and its service adapters.

SIB:

The *SIB* is the (graphical) building block used by the application expert for composing models in jABC. As pointed out above, it is an abstract representation of a specific behavior or functionality, providing parameters for configuration and branches for reflecting the possible execution results. Technically, such a SIB is described by means of a very simple Java class which defines the SIB's constituents via programming conventions:

- Parameters are defined by all public fields of the Java class. All parameters have to be initialized with default values (i.e. in particular `null` values are not allowed).
- Branches are separated into *final* and *mutable branches*. Final branches cannot be modified by the application expert when configuring the SIB, whereas mutable branches can be renamed or deleted. The latter may be useful if one or more execution results emerge dynamically when the represented service is executed. The final branches of a SIB are defined by a final static String array called `BRANCHES`, and the mutable branches are specified in a non-constant String array named `branches`. Furthermore, if the `branches` array is initialized at least as an empty array, this enables the possibility for the application expert to add new mutable branches via the jABC tool.
- An icon and the SIB's documentation are specified by implementing special methods.
- Optionally, plugins may introduce further information or functionality by means of corresponding interfaces, which then can be implemented by the IT expert (see Sect. 3.2.3 and 3.3.1 for examples).

Finally, the class is marked as a SIB via an annotation (`@SIBClass`), which also declares a *unique identifier* (UID) in order to reliably distinguish the SIB from other SIBs.

The jABC framework only allows a restricted set of data types for specifying a SIB's parameters. The set is separated into *simple* and *complex types*. The simple types consist of standard Java data types such as `Boolean`, `String`, numeric types (e.g., `Integer`, `Float`), `File`, arrays and various collections (e.g., `ArrayList`, `HashMap`).

Furthermore, the framework supports several complex types which are listed in Table 3.1. The data types written in italics have been designed for very specific application scenarios and are rarely used, thus they will not be considered further in this book. The remaining data types are used more commonly and serve different purposes. Some of those complex types represent data that allows to deal with jABC-specific concepts. For instance, `ContextExpression` and `ContextKey` enable working with contents of the

Table 3.1. Complex built-in data types in jABC

Complex Data Type	Represented data
ContextExpression	EL expression evaluated on the execution context (cf. Sect. 3.3.2)
ContextKey	Key for accessing contents of a particular execution context (cf. Sect. 3.3.2)
ExtendedFile	File located relative to the current jABC project, may be restricted to specific file types
<i>JavaBeanReference</i>	Arbitrary Java object following the JavaBeans programming model [Ora11d]
ListBox	Single object/item selected from a fixed list of values
MultiObject	Aggregated object, similar to a record in Pascal or a struct in C
<i>ObjectReference</i>	Arbitrary object
Password	Password string
<i>SIBLink</i>	Link to another SIB instance in an arbitrary SLG
StrictCollection	Typed collection
StrictList	Typed list
<i>Variable</i>	Typed variable

execution context (Sect. 3.3.2 elaborates on the usage of those types). Further complex types mainly have the purpose of indicating the display of particular user interface elements in the jABC tool, in order to adequately support the application expert in specifying the corresponding data. Accordingly, the type `ListBox` signals the use of a combo box that allows the selection of one value from a fixed list of items, and the type `Password` leads to a typical password input field which only shows asterisks instead of the entered text. Finally, other complex types compensate the lack of particular data types or features in Java, such as `MultiObject`, which resembles an aggregated object similar to records in Pascal. Furthermore, `StrictCollection` and `StrictList` represent typed collections which, in contrast to Java's generics that are subject to type erasure [Gos+05, p. 56], also retain and use their type restriction at runtime. As another convention in jABC, all collection and array types must only contain values which are in turn assignable to one of the supported simple or complex types.

Please note that this set of data types represents the *technical* grounding for realizing the parameters of a SIB. Accordingly, those data types are typically only visible to the IT expert who implements the SIB. The application expert who uses a SIB in the jABC tool usually only deals with abstract types. During the customization described above, the domain expert defines such abstract types and maps them to the corresponding technical data types [NLS11].

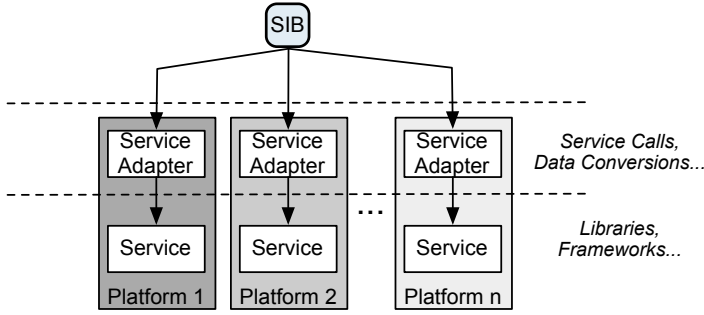


Fig. 3.3. Service adapter pattern for realizing a SIB's behavior

Service Adapters:

A *service adapter* realizes the SIB's behavior for a concrete target platform using the object adapter pattern [Gam+95]. Particularly, as one SIB may be executable on multiple target platforms, an arbitrary number of service adapters can be attached to a SIB. Fig. 3.3 illustrates this pattern. Each service adapter is implemented in a programming language supported by the desired target platform, so it may for instance be a Java class, a C# class or a Python script. Typically, it contains calls to, e.g., platform-specific third party libraries or systems, that realize the actual service represented by the SIB, along with corresponding data conversions. Decoupling the concrete platform-specific implementations from the SIB description assures that the SIB itself is entirely platform-independent. As soon as at least one service adapter is implemented for a SIB, the SIB is executable, thus enabling interpretation and code generation for models that contain the SIB. Sect. 5.2.1 elaborates on the high importance of the service adapter concept for code generation in jABC.

3.2.2 Service Logic Graphs

As mentioned above, the models in jABC are called Service Logic Graphs (SLGs). Basically, SLGs are directed graphs that represent the flow of actions in an application, thus focussing on its behavioral (dynamic) aspects [MS06]. In formal terms, SLGs are *Kripke Transition Systems* (KTS) [MOSS99; MS09a], a combination of Kripke structures and labeled transition systems. Accordingly, nodes as well as edges are labeled in a KTS:

Definition 1 (Kripke Transition System, KTS). A KTS over a set of atomic propositions AP is a four-tuple $\mathcal{M} = (S, \mathcal{A}, \rightarrow, I)$ where

- S is a finite set of states (nodes),
- \mathcal{A} is a finite set of action labels,

- the transition relation $\rightarrow \subseteq S \times \mathcal{A} \times S$ describes possible action-triggered transitions between states, and
- the interpretation function $I : S \rightarrow 2^{AP}$ specifies which atomic propositions hold at which node.

AP is assumed to always contain the propositions **true** and **false**, and for any state $s \in S$, **true** $\in I(s)$ and **false** $\notin I(s)$. A finite path π in \mathcal{M} is considered a sequence of states and action labels $\pi = \langle s_1, a_1, s_2, a_2, s_3, \dots, s_n \rangle$ with $s_i \in S$ and $a_i \in \mathcal{A}$, such that $(s_i, a_i, s_{i+1}) \in \rightarrow$ (also written $s_i \xrightarrow{a_i} s_{i+1}$) for $i = 1, \dots, n-1$. Infinite paths are defined in a similar manner [MOSS99]. Furthermore, π_i refers to the state s_i on the path.

The set of atomic propositions AP and the interpretation function I are mostly required for verification, which is described in more detail in Sect. 3.4. In the context of SLGs, the nodes in such a graph are SIB instances or, in order to enable hierarchical modeling (cf. Sect. 3.1), *macros* that point to other SLGs. For instance, in the example model depicted on the top of Fig. 3.4, the nodes labeled **Print Exception** and **Print Success** are SIB instances, while the nodes labeled **Initialize Docu Generator** and **Generate Documentation** are macros (indicated by the big dot on their icons).

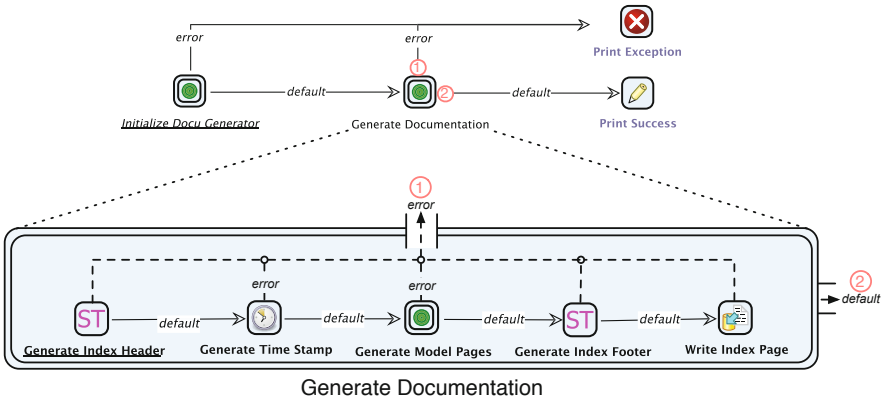


Fig. 3.4. Hierarchical models in jABC

The directed edges between the nodes, described by the transition relation \rightarrow , indicate the flow of actions. In SLGs, the actions connecting two nodes are reflected by branches: Each edge is labeled with one or more branches, whereas the source node of the edge defines the set of possible branches that can be assigned to that edge. In other words, the wiring of SIB instances and macros in models is performed on the basis of possible execution results. Thus roughly speaking, branches could also be seen as potential “exits” of a SIB/macro.

If a node has more than one outgoing edge, the edges represent alternative execution flows. For instance, the node **Initialize Docu Generator** in Fig. 3.4 has two branches “default” and “error”, each assigned to one outgoing edge. This reads as follows: *If the result of **Initialize Docu Generator** is “default” proceed with **Generate Documentation**, if the result is “error” execute **Print Exception**.* If **Initialize Docu Generator** produces another result, it is considered an undefined behavior. In order to specify where the execution of a model starts, a node can be defined as an entry point. This is indicated by the node’s label being underlined (e.g., **Initialize Docu Generator** in Fig. 3.4). Please note that an SLG could potentially have more than one entry point, depending on whether this is supported by the selected interpreter or code generator.

Hierarchical Modeling:

In order to enable seamless hierarchical modeling, macros are used just like normal SIBs. For the application expert, the SLG referenced by a macro is also considered a service (with the difference that its realization is available as another model) that follows the same simple interface that is also imposed on SIBs. In consequence, macros also have parameters and branches. However, as these parameters and branches belong to an entire model associated with the macro, they are called *model parameters* and *model branches*, respectively.

An SLG’s set of model parameters and model branches is defined by selectively *exporting* parameters and branches of SIB instances or macros in that SLG. Fig. 3.4 illustrates this for model branches. The bottom part of the figure shows the submodel that is associated with the macro **Generate Documentation**. This submodel has again one designated entry point, as indicated by the underlined label of **Generate Index Header**. For proper execution semantics, we also need to specify at which points the submodel can be left in order to return to the parent model. This is done by means of model branches, which, analogous to SIB branches, define the “exits” of models. In the jABC tool, such exits are not visualized by concrete edges pointing to the parent model, but instead the information is displayed in one of the inspectors (the *Graph inspector*). For illustration, Fig. 3.4 indicates the model exits via dashed arrows.

In the example, each SIB instance contained in the submodel has a branch labeled “error”, all of them exported and mapped to a model branch which is also called “error” ①. The name of a model branch or model parameter can be defined freely by the application expert. Resulting from this specification of the “error” model branch, the error handling for all execution steps in the submodel is delegated to the parent model. Furthermore, the SIB instance **Write Index Page** exports its “default” branch as a model branch which is also called “default” ②. In the parent model, the macro **Generate Documentation** provides exactly those two exits “default” and “error”, which are defined as model branches in the underlying submodel. As with normal SIBs, these branches then can be assigned to outgoing edges of the macro.

Likewise, it is possible to define model parameters of a submodel, which then become the parameters of an associated macro.

Technical Realization:

Technically, jABC manages SLGs by means of a data structure called **SIB-GraphModel** which strongly focuses on robustness. In particular, the data structure ensures that SLGs can always be opened and modified, even if some or all of the contained SIBs are not available (e.g., because a particular SIB bundle has not been installed). For this purpose, any affected SIBs are replaced by a *proxy SIB*, which is a special generic SIB that is able to emulate any other SIB's interface. Thus the proxy SIB that replaces another SIB has the same parameters and branches, and both can be manipulated in the jABC tool as usual. Consequently, it is entirely transparent to the application expert whether all SIBs contained in his models are actually available.

While being an adequate replacement at modeling time, proxy SIBs are not able to emulate the runtime behavior of a SIB. As proxy SIBs lack corresponding service adapters, they are not executable. Thus proxy SIBs can only compensate the effects of missing SIBs in a way that the application expert's work is not interrupted – however, as missing SIBs threaten the executability of the modeled system, they pose a problem which has to be fixed by the IT expert.

Besides its focus on robustness, jABC's **SIBGraphModel** data structure is very extensible. For instance, plugins are allowed to attach arbitrary information, called *user objects*, to all constituent parts of a model (i.e., nodes, edges and the model itself). Similar to a hash table, any user object associated with a model element is identified by a unique key.

Metamodeling:

From the metamodeling perspective, SLGs and their associated concepts (SIBs, branches etc.) are jABC's metamodel. As this metamodel is hard-wired in the framework, it is not interchangeable and thus cannot be substituted by an entirely different one, i.e., jABC is not a language workbench in the sense described in Sect. 2.3.5. In terms of the metalevels proposed by the OMG (cf. Sect 2.3.3), modeling in jABC mostly happens on level M1.

However, this is a deliberate design decision that aims at reducing the framework's overall complexity in favor of simplicity. In contrast to CASE tools with their fixed modeling languages (cf. Sect. 2.3.1), jABC's metamodel is not static, as it can be customized in a number of ways. For instance, its abstract syntax can be dynamically extended by new SIBs which thus increase the size and expressibility of the modeling language. Additionally, the domain expert's customization possibilities outlined above allow tailoring the modeling language to particular domains, e.g., by adjusting the available language elements via taxonomies, or by adapting the concrete syntax of SLGs (terminology, icons etc.). By means of a special class of SIBs called

“control SIBs” (cf. Sect. 3.3.3) it is even possible to introduce entirely new modeling constructs. However, such new constructs usually also require a corresponding adaptation of jABC’s tooling (e.g., plugins). In this respect, the modeling languages derived from jABC’s metamodel show characteristics of both internal (addition of new SIBs, customization by the domain expert) and external (addition of new modeling constructs) DSLs (cf. Sect. 2.2).

Apart from the abstract syntax, the static semantics of jABC’s metamodel can also be customized via







- local constraints (cf. Sect. 6.1) which are attached to each SIB,
- global constraints (cf. Sect. 6.2) which are (domain-specifically) defined for models, and
- plugins (cf. Sect. 3.2.3) which may, e.g., introduce additional well-formedness rules.

This clearly contrasts the approach of language workbenches, which usually generate a domain-specific environment from a metamodel specification. jABC achieves this domain-specificity by means of customization and adaptation of a generic metamodel and tool. This idea is comparable to UML’s profile mechanism (cf. Sect. 8.1).

In summary, with this *customizable and extensible metamodel*, jABC defines a *class of domain-specific languages* that share a fixed (and thus controlled) common core, the SLG concept.

SLG Types:

In this book, SLGs are used for many different purposes. Hence in the following chapters and sections, any figure containing an SLG is marked with a small icon in one of its corners, in order to avoid confusion. The icons indicate the type of the depicted model:

 Code generator	 Any application
 Formula	 Test case
 Test suite	 Test data

3.2.3 Plugins

Plugins allow adding functionality to jABC, e.g., by extending the jABC tool with further menus or inspectors, or by enriching SLGs and their constituents with additional information (i.e., user objects). Plugins are even able to associate different semantics with one SLG. While the previous section already anticipated the most common semantics according to which an SLG is a control flow graph, there are also plugins that interpret SLGs differently, e.g., as an entity-relationship model [Win06] or even as a formula (cf. Sect. 6.2.1). Like jABC itself, plugins are implemented in Java and then integrated in jABC via a simple interface.

As mentioned above, the selection of suitable plugins is an important task of the domain expert who customizes jABC for a particular application domain. For this purpose, a multitude of plugins is available, supporting different aspects of system development with jABC. Two plugins that are very important from the perspective of this book are presented in detail in separate sections: the *Tracer*, which allows the execution of SLGs (Sect. 3.3), and *GEAR*, which enables SLG verification via model checking (Sect. 3.4). Furthermore, the Genesys framework also provides a jABC plugin in order to support code generation for SLGs (cf. Sect. 4.3). The *FormulaBuilder*, another plugin that extends jABC by the ability of modeling and generating formulas, has been realized on the basis of Genesys and thus is also presented later on in Sect. 6.2.1. Further relevant plugins are briefly introduced below.

LocalChecker:

The *LocalChecker* plugin [Neu07; Ste+07] ensures the correct use of SIB in models by checking local constraints (cf. Sect. 3.1) that are attached to each SIB. The checks that realize those constraints are directly implemented as Java code: The IT expert adds them to the SIB's Java class (see Sect. 3.2.1) by implementing a special interface (*LocalCheck*). For convenience, the *LocalChecker* provides a set of general standard checks that are domain-independent and thus can be used for most SIBs. The bulk of those checks concern the well-formedness of the SLGs (i.e., they check an SLG's conformance with the static semantics specified by the metamodel, cf. Sect. 2.2). Examples of such checks include the correct parametrization of a SIB (valid codomain, input syntax etc.), branches that are not assigned to any outgoing edges, or unconnected edges that do not have a valid source or target node. The standard checks already cover the most common modeling mistakes.

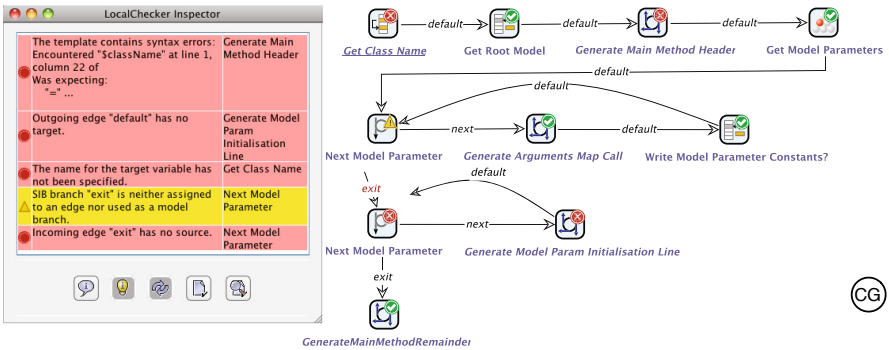


Fig. 3.5. The *LocalChecker* plugin in jABC [JMS11]

While modeling in jABC, all checks specified for the employed SIBs are performed continuously. Any feedback obtained from the checks is immediately reported to the user via an additional inspector, which is depicted on the

left side of Fig. 3.5. The figure also shows several examples of messages produced by different checks, and it is visible that those messages are classified by severity as errors, warnings or plain informations. For instance, the SIB “Next Model Parameter” causes a warning (second line from bottom in the inspector), and for a second instance of the SIB, an incoming edge without a proper source node leads to an error message (last line in the inspector). Besides the messages in the inspector, the check results are also indicated directly in the SLG on the icon of each SIB. Sect. 6.1 elaborates on how the LocalChecker has been employed in the context of this book.

Annotation Editor:

The *Annotation Editor* [Nag09] allows attaching almost any kind of information to jABC projects as well as to SLGs and their constituent parts (such as SIB instances or edges). By means of grammars (which are also specified as SLGs), the Annotation Editor can be tailored to specific application scenarios. Such a grammar determines which kind of information is allowed to be attached to which elements. From the grammar, the Annotation Editor dynamically assembles a correspondingly adapted user interface for creating and editing suitable information in jABC. An important application of the Annotation Editor is, e.g., the documentation of SLGs and their contained elements.

Taxonomy Editor:

This plugin is mainly used by the domain expert when customizing jABC for a particular application domain. It allows naming and organizing the available SIBs in a way that they fit the terminology and concepts of the targeted domain. For this purpose, the Taxonomy Editor provides a user interface that shows all available SIBs listed according to their physical Java package structure. Starting from this view, the domain expert renames and rearranges the SIBs as required.

jETI plugin:

Java Electronic Tool Integration (jETI) [SMN05] augments jABC by enabling the inclusion of *remotely* available services and tools as SIBs. Such SIBs can be used just like any other SIBs, i.e., the communication with the remote services is seamless and transparent to the application expert. The corresponding remote services or tools either have to be available on the internet in some standardized form (e.g., as Web Services [Pap08]), or they are provided by means of jETI’s own tool server. In both cases, jETI is able to automatically generate the corresponding SIBs [SMN05;Kub+09], that allow the application expert to integrate those remote services or tools in his SLGs.

3.3 Model Execution with the Tracer

The *Tracer* [Doe06; Ste+07; JMS08] enables the direct execution of models and can thus be considered an interpreter for SLGs. As such it is key to activities such as rapid prototyping, debugging and monitoring. The Tracer is separated into two parts: a general execution environment for SLGs and a corresponding jABC plugin.

The *execution environment* is an integral part of the jABC framework, and it provides

- an *execution semantics* for SLGs, incarnated by a corresponding interpreter (thus being an example of pragmatic semantics, cf. Sect. 2.3.6),
- the concept of the *execution context* (already mentioned in Sect. 3.2.1) which acts as a shared memory for SIB instances, as well as
- an interface for adding new control flow mechanisms via specific SIBs called *control SIBs*.

The *Tracer plugin* augments the jABC tool with facilities for starting, controlling, observing and debugging SLG executions.

The Tracer is particularly essential to the realization of the Genesys approach. First, any code generator modeled with Genesys (and thus in jABC) can be, just like any other SLG, immediately tested by executing it with the Tracer. Second, when using Genesys for building code generators for jABC (cf. Sect. 5), the Tracer is the technical basis for an entire class of generators called *Extruders* (see Sect. 5.1). Finally, the execution semantics defined by the Tracer is a guideline for the behavior of the generated code, which should coincide with the behavior of the traced SLG (*execution equivalence*, see Sect. 5.1).

The following sections elaborate on the constituent parts of the Tracer.

3.3.1 Execution Semantics

In parts, the execution semantics for SLGs which is used by the Tracer has already been anticipated above in Sect. 3.2.2. The execution of an SLG always starts with one designated entry point (also called *start SIB*). When a SIB instance is reached by the interpreter, the underlying service is executed using the configuration that results from the SIB's parameter values. How this happens exactly is specified by the IT expert: In order to be executable by the Tracer, the SIB's Java class (see Sect. 3.2.1) has to implement the Tracer interface **Executable**. This implementation describes the SIB's behavior when it is executed by the interpreter. For this purpose, it may call a particular service or service adapter, it may directly implement the service itself (though this has several disadvantages for code generation, see Sect. 5.2.1), or it may contain mock code or delegate to a mock service. The latter is an option for

enabling executability even if the actual service represented by a SIB is not available yet. If the reached SLG node is not an instance of a SIB but a macro, the execution descends to the referenced submodel and proceeds with the submodel's entry point. The submodel is configured with the values specified for the macro's parameters.

By all means, the result of executing a SIB instance or macro should always be one of the available branches (i.e., branches defined for the SIB or macro). The interpreter then determines the outgoing edge that is labeled with this branch and continues executing the corresponding successor. If the result reflects a model branch, i.e., an “exit” of the SLG, the execution proceeds with the parent model, and searches for a suitable outgoing edge at the macro which previously caused the interpreter to descend the hierarchy. Correspondingly, in this execution semantics, multiple outgoing branches of a SIB instance or macro usually represent alternative execution flows (though this pattern may be changed, see “Control SIBs” below).

The execution of an SLG is finished regularly (i.e., successfully) as soon as it reaches a SIB or macro that is contained in the topmost model of the hierarchy and that does not have any outgoing edges. The following situations may cause an execution to stop irregularly:

- The execution does not end at the topmost hierarchy level.
- The execution of a SIB instance produces a result which is not among the branches defined for the SIB.
- The execution of a SIB instance fails (e.g., because it has been replaced by a proxy SIB, see Sect. 3.2.2).
- An SLG defines multiple entry points or none at all.

In those cases, the Tracer aborts the execution and reports an error to the user.

3.3.2 Execution Context

As mentioned above, the execution context is a shared memory that enables communication of SIB instances in a model. Any SIB instance is able to put data into the execution context and to read, edit and delete data stored by other SIB instances. Similar to a hash table, each datum contained in the execution context is identified by a unique identifier called *context key*.

In order to enable modeling recursion with an SLG hierarchy, the Tracer also allows stacked execution contexts. In this setting, each SLG in the hierarchy is associated with its own execution context, similar to a local namespace. Each time the execution descends to a submodel referenced by a macro, a new execution context is put on the stack of contexts, and when the execution of the model is finished, the context is removed from the stack and thus ceases to exist. This concept allows shadowing of context keys, which enables full

support for recursion as known from programming languages. A recursive SLG can, e.g., be constructed by means of a macro which in turn references the SLG in which it is contained. The application expert takes the decision whether a submodel should be executed with its own local execution context by selecting a corresponding type of macro (see Sect. 3.3.3). Consequently, stacking of execution contexts can be flexibly enabled or disabled for each hierarchy level.

Independent of execution context stacking being enabled or not, the Tracer always provides one designated global execution context which is accessible from anywhere in the SLG hierarchy. This global context is even maintained after the actual execution has been terminated, e.g., in order to avoid the loss of data that resulted from the SLG execution.

Besides being able to work with this global execution context and an optional local execution context associated with their model, SIB instances are also allowed to access any other context that may be present on the stack. For identifying the particular execution context that should be accessed, the Tracer defines four *scopes*:

- *global* designates the global context,
- *local* is the local context associated with the currently executed SLG,
- *parent* refers to the superordinate context on the stack, which usually belongs to the current SLG's parent model, and
- *declared* references the first context on the stack that contains a specific key (starting from the local context).

When working with execution contexts, SIB instances that are supposed to share data have to agree on the corresponding keys. Accordingly, SIBs may provide parameters of type **ContextKey** (see Sect. 3.2.1) which allow the application expert to configure how the execution contexts are accessed by a SIB. For this purpose, such a **ContextKey** parameter demands the specification of a name for the key that identifies the data in the context, as well as the selection of a scope for determining which context should be accessed. For instance, the SIB inspector depicted in Fig. 3.2 (3) in Sect. 3.2 shows an example of a context key parameter called “result” which specifies the key “indexPage” along with the scope “global”. Most SIBs in jABC's ready-made libraries (such as the Common SIBs) provide this configurability.

Another type of SIB parameter for working with contents of the execution contexts is **ContextExpression** (see Sect. 3.2.1). This data type uses *context expressions* written in the expression language (EL) introduced with the JavaServer Pages Standard Tag Library (JSTL) [Jav06] in order to enable *dynamic* access to the execution contexts. The Tracer's execution environment provides corresponding resolvers for evaluating such expressions during the execution. For instance, the simple expression `${keyName}` is

resolved to the value identified by the context key `keyName`. In this expression, anything between the characters `${` and `}` is interpreted as a context key, and upon evaluation the resolvers search the correspondingly referenced value in the execution contexts using the scope “declared”. Furthermore, context expressions can be used to perform more complex tasks such as concatenating character data (e.g., `${key1}${key2}`), accessing attributes of objects stored in the context (e.g., `${keyOfObject.attributeName}`), performing comparisons (e.g., `${key1 <= key2}`) or calling external functions (e.g., `${key}_${Math.random()}` for suffixing the string value of `key` with an underscore and a random number). In consequence, as expressions may contain static strings along with the dynamically evaluated parts (such as `_` in the last example), context expressions may even be used as a simple template language² (cf. Sect. 2.4.2). However, as working with context expressions requires certain technical skills, they are mostly intended to be used by IT experts on the lower levels of an SLG hierarchy, rather than by application experts.

As to the experience of the author, advanced features like stacked execution contexts or context expressions are not required in most application scenarios. Especially for applications which do not require recursion, resorting to one single (i.e., *flat*) execution context instead of the stack variant is often preferable. This way, application experts do not have to consider scopes at all, which noticeably eases the creation of SLGs. Furthermore, for application experts, context expressions complicate the usage of corresponding SIBs as they require to learn and to apply the expression language. Hence instead of producing generic, highly configurable SIBs with context expressions, the development of more domain-specific SIBs is preferable, because such SIBs are usually significantly easier to use.

For the sake of simplicity, the remainder of this book will use the singular notion “the execution context” for both a flat execution context and for the stack variant.

3.3.3 Control SIBs

As a further important feature, the Tracer’s execution environment supports the extension and adaptation of the standard execution semantics for SLGs described above. Such extensions can be performed by means of specific SIBs called *control SIBs*. In contrast to normal SIBs, control SIBs do not represent an underlying service that is executed by the Tracer. Instead, when the execution arrives at a control SIB, the Tracer completely hands over the execution control. As control SIBs are granted full access to the execution environment,

² In fact, the EL language combined with a corresponding implementation, such as the resolvers provided by the Tracer’s execution environment, is just another template engine.

they are, e.g., able to create new execution contexts or even new execution threads. Given these competences, control SIBs are able to alter existing control flow patterns and to define new ones. Currently, jABC provides a small set of standard control SIBs that add control flow patterns for hierarchy, multi-threaded execution as well as event handling.

The previous sections already stated that *hierarchy* in SLGs is created by means of macros. In fact, those macros are control SIBs which tell the Tracer to continue the execution with corresponding submodels. As mentioned above in the description of the execution context, there are several types of macros that represent different ways for executing a submodel. The standard macro is the **MacroSIB**, which simply executes the referenced submodel using a flat execution context. In contrast to this, the **GraphSIB** creates a new local context for the execution of the submodel (i.e., stacked execution contexts are used). The **ThreadSIB** behaves similarly, but it additionally executes the submodel in a separate thread. Thus by selecting one of those three macros, the application is able to determine how the referenced submodel should be executed and in particular which type of execution context (flat or stacked, cf. “Execution Context” above) is employed.

Further control SIBs allow the use of *multi-threading* in SLGs. For this purpose, the **ForkSIB** modifies the standard execution semantics that usually treats multiple outgoing edges as alternative execution paths. For the **ForkSIB**, each outgoing edge represents a separate thread, so that upon execution, the Tracer follows all specified outgoing edges in parallel. Via another control SIB, the **JoinSIB**, threads can be synchronized and remerged as one single thread. SLG 4 in Fig. 6.9 (Sect. 6.3.1) illustrates the use of those two control SIBs. When executing this model, the **ForkSIB** creates one thread for each outgoing edge (“Thread1” and “Thread2”), so that the macros **Sequence** and **Recursion** are executed in parallel. Afterwards, the **JoinSIB** synchronizes the two threads, i.e., it waits for both threads to finish and then proceeds with the execution in one single thread.

In order to enable proper execution, the Tracer defines strict rules for the well-formedness of such fork-join constructs. First, fork and join are only allowed to occur pairwise, i.e., for each **ForkSIB** there has to be exactly one corresponding **JoinSIB** that synchronizes exactly those threads created by the **ForkSIB**. Correspondingly, the number of outgoing edges of a **ForkSIB** (i.e., the number of created threads) has to match the number of incoming edges of the corresponding **JoinSIB**. In particular, this entails that the modeled threads between the **ForkSIB** and the **JoinSIB** must not contain any “exits”, such as model branches or other paths that bypass the **JoinSIB**. Furthermore, fork-join constructs have to be nested in a way that does not violate any of the above rules. If a fork-join construct violates one or more of these rules it is not well-formed and thus will lead to an irregular termination of the execution.

Finally, there are also control SIBs that support event handling. Those control SIBs allow to suspend the execution of a model until it receives a specific event. Furthermore, a submodel can be specified that handles the incoming event. In order to avoid infinite waiting, the suspension of the execution can be restricted by means of a timeout.

The control flow mechanisms supported by the standard execution semantics and by the existing control SIBs have proven sufficient for all application scenarios of jABC so far. However, further control flow patterns (e.g., those from [VDA+03] that are not yet supported) can be added on demand by implementing corresponding control SIBs.

3.3.4 Tracer Plugin

The *Tracer plugin* can be considered a debugger for SLGs . Besides allowing to simply execute an SLG straightaway by pushing a “play” button, it is also able to graphically visualize the current state of an execution. This is depicted in Fig. 3.6. In the SLG on the right hand side, the SIB that is currently executed (**Generate Model Pages**) is marked by a little icon, and the previous path of the execution is also made visible by means of highlighted edges.

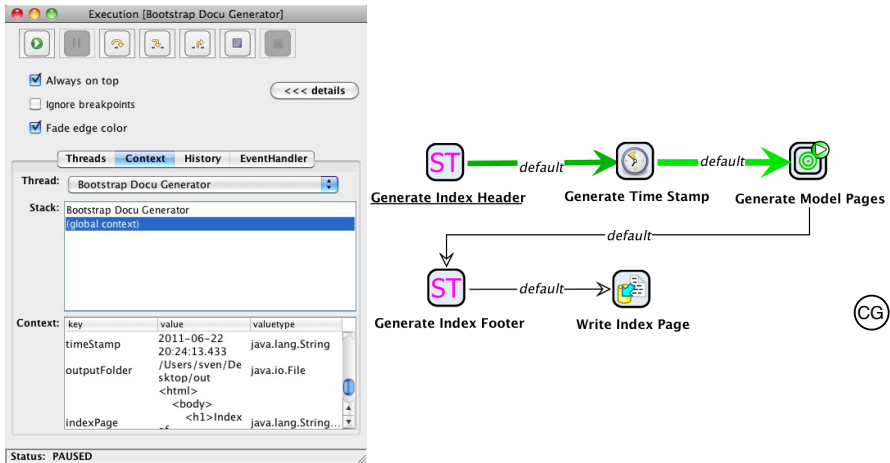


Fig. 3.6. Executing an SLG with the Tracer

The dialog that is shown on the left hand side of Fig. 3.6 has two purposes. First, it allows to control the execution: By means of the buttons on the top, the user is able to start, pause and stop the execution at any point. Second, the dialog displays information about the current execution, such as the currently

running threads, the current contents of the execution contexts and a history reflecting the execution path so far.

Instead of visually executing the SLG step by step, the user may also define breakpoints which cause the execution to stop at specified points of interest.

3.4 Model Checking with GEAR

As outlined in Sect. 3.1, safeguarding the consistency of the global system model while it is composed from models and services is a central requirement of the XMDD approach, as it helps overcoming typical system gaps. Furthermore, only a consistent global system model can be properly executed and translated to working code. The rules that have to be followed (or, in other words, the properties that have to be satisfied) in order to ensure this consistency are specified by means of (temporal) constraints, which are an integral part of XMDD's model library. Those constraints form a steadily growing library and thus are the basis of incremental formalization (cf. Sect. 3.1). Sect. 3.2.3 already presented the LocalChecker plugin, which focusses on verifying local constraints that concern the single SIB instances in a model. However, an additional check tool is required that takes care of global constraints which describe the rules addressing entire models.

As also stated above, a central advantage of shifting all activities of system development to the modeling level is the fact that models are amenable to formal methods. Among those, *model checking* [CGP99; QS82; MOSS99] is an established technique for checking whether a given model satisfies a specified property. The *GEAR plugin* [Bak+09] enables user-friendly model checking of global constraints for SLGs in jABC based on a game-based approach [Bak+07; MOY04]. The following sections introduce this plugin from the user perspective: Sect. 3.4.1 elaborates on how global constraints for GEAR are specified and Sect. 3.4.2 describes how the GEAR model checker is actually used via the corresponding jABC plugin. Please note that the sections only focus on those parts of GEAR which are important in the context of this book. For more details on GEAR's conceptual and algorithmic aspects, in particular on the realization and impact of the employed game-based approach, please refer to [Bak+07; MOY04; Yoo07]. Finally, Sect. 6.2 elaborates on how GEAR has been applied for model checking of code generators.

3.4.1 Specification of Global Constraints

When working with model checkers, constraints or properties that should be verified are typically specified by means of temporal logics [CGP99]. This approach is quite general, as temporal logic is sufficient to express any required

property [Ste89; SI94]. GEAR essentially is a model checker for the *modal μ -calculus* [Koz83], which is its core logic for formulating global constraints. However, GEAR also supports “derived, more user-friendly logics” [Bak+09] (or also “domain-specific specification languages” [Bak+09]) such as the Computation Tree Logic (CTL) [BAPM83; CE81]. Such alternative logics and specification formalisms can be flexibly added to GEAR on the basis of so-called *macros* (called *GEAR macros* in the following in order to avoid confusion with the macros used in SLGs for modeling hierarchy, cf. Sect. 3.2.2). Basically, a GEAR macro is an abbreviation or a pattern that represents a specific formula, and that can be used like a simple black box (similar to the idea of SIBs), thus leading to more readable and concise constraints. When using a GEAR macro, GEAR automatically expands it and internally – i.e., transparently to the user – translates it to the modal μ -calculus. This even allows mixing several formalisms and languages in one constraint specification (see Sect. 6.2 for example constraints).

The flexibility and extensibility of GEAR’s input syntax is a powerful feature as it allows to continuously improve the accessibility and simplicity of property specifications. For instance, this mechanism is used to incorporate the property specification patterns proposed by Dwyer et al. [DAC99]. Those patterns capture common constraints concerned with the occurrence or order of actions in a formalism-independent way, and thus are, in terms of their abstraction level, on par with GEAR macros. Sect. 6.2 shows several examples of those patterns, which are frequently used in this book for specifying constraints for code generators.

The FormulaBuilder is an extension of GEAR that aims at further easing the specification of constraints. This jABC plugin allows to graphically formulate constraints as SLGs, which can be seamlessly used, just like any other constraints, for performing model verification with GEAR. Since the FormulaBuilder is also an application of the Genesys approach, it is presented in detail in Sect. 6.2.1.

Finally, as this book uses a variant of CTL for the formal description of constraints, this logic is briefly introduced in the following.

Computation Tree Logic (CTL)

CTL can be used to formulate temporal constraints of a model or, more formally, of a computation tree, which can be represented, e.g., as a Kripke structure [CGP99] or as a Kripke Transition System (cf. Sect. 3.2.2). For instance, such constraints are concerned with the reachability or order of certain states. For the specification of constraints, CTL employs path quantifiers and temporal operators [CGP99]. The former refer to the branching structure of a model, and the latter describe properties that hold on a single path. The

possible path quantifiers in CTL are A meaning “for all paths”, and E meaning “for some paths”. Furthermore, there are four temporal operators:

- $X \phi$ (“next”): The property ϕ holds in the next state of the path.
- $G \phi$ (“globally”): ϕ holds in all states of the path.
- $F \phi$ (“finally”): ϕ holds eventually at some state of the path.
- $\phi U \psi$ (“until”): ψ holds at some state of the path and ϕ holds in all preceding states. This operator is sometimes also called “strong until” as it requires ψ to hold finally [MOSS99]. There is also a commonly used variant of this operator called “weak until” ($\phi WU \psi$) which does not demand the occurrence of ψ , but instead also allows ϕ to hold forever.

Due to the fact that its operators quantify over paths that start in a specific state of the model, CTL belongs to the *branching-time logics* [MOSS99]. For the formal description of temporal constraints, this book uses a variant of CTL based on a logic described by Schmidt and Steffen [SS98]. This variant adds the box ($[]$) and diamond ($\langle \rangle$) operators known from Hennessy-Milner logic [HM85; Mil89] along with corresponding backward counterparts ($[\bar{}]$ resp. $\langle \bar{} \rangle$). Those operators are basically parametrized versions of the “next” operator that allow to refer to certain actions. The following definition describes how formulas of the CTL variant are composed syntactically:

Definition 2 (Syntax of CTL variant, based on [SS98] and [MOSS99]).

The syntax of a CTL formula is defined as follows³:

$$\begin{aligned} \phi ::= & p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \\ & \mid [a] \phi \mid [\bar{a}] \phi \mid \langle a \rangle \phi \mid \langle \bar{a} \rangle \phi \\ & \mid AG \phi \mid EG \phi \mid AF \phi \mid EF \phi \mid A[\phi U \phi] \mid E[\phi U \phi] \end{aligned}$$

with p being an element of a set of atomic propositions AP and a being an element of a finite set of action labels \mathcal{A} .

Please note that the definition is reduced to those syntactic elements that are required for the descriptions in this book. Consequently, it does not contain all CTL extensions introduced in [SS98] (e.g., the parametrized version of the AG operator).

The semantics of those formulas can be defined with respect to Kripke Transition Systems (KTS, see Definition 1), which are the formal basis of SLGs :

Definition 3 (Semantics of the CTL variant, based on [HR04, p. 211] and [MOSS99]). *Let $\mathcal{M} = (S, \mathcal{A}, \rightarrow, I)$ be a KTS. The relation $\mathcal{M}, s \models \phi$ (read as “ ϕ holds in state s of \mathcal{M} ”) is defined by structural induction on ϕ :*

³ The notation of the syntax description is based on the well-known Backus-Naur Form (BNF) [Bac+63].

1. $\mathcal{M}, s \models p \Leftrightarrow p \in I(s)$
2. $\mathcal{M}, s \models \neg\phi \Leftrightarrow \mathcal{M}, s \not\models \phi$
3. $\mathcal{M}, s \models \phi_1 \wedge \phi_2 \Leftrightarrow \mathcal{M}, s \models \phi_1 \text{ and } \mathcal{M}, s \models \phi_2$
4. $\mathcal{M}, s \models \phi_1 \vee \phi_2 \Leftrightarrow \mathcal{M}, s \models \phi_1 \text{ or } \mathcal{M}, s \models \phi_2$
5. $\mathcal{M}, s \models \phi_1 \Rightarrow \phi_2 \Leftrightarrow \mathcal{M}, s \not\models \phi_1 \text{ or } \mathcal{M}, s \models \phi_2$
6. $\mathcal{M}, s \models [a] \phi \Leftrightarrow \text{for all } t \text{ with } s \xrightarrow{a} t \text{ we have } \mathcal{M}, t \models \phi$
7. $\mathcal{M}, s \models \overline{[a]} \phi \Leftrightarrow \text{for all } t \text{ with } t \xrightarrow{a} s \text{ we have } \mathcal{M}, t \models \phi$
8. $\mathcal{M}, s \models \langle a \rangle \phi \Leftrightarrow \text{there exists a } t \text{ with } s \xrightarrow{a} t \text{ such that } \mathcal{M}, t \models \phi$
9. $\mathcal{M}, s \models \overline{\langle a \rangle} \phi \Leftrightarrow \text{there exists a } t \text{ with } t \xrightarrow{a} s \text{ such that } \mathcal{M}, t \models \phi$
10. $\mathcal{M}, s \models AG \phi \Leftrightarrow \text{for all paths } \pi \text{ with } \pi_1 = s \text{ and all } s_i \text{ along the path, we have } \mathcal{M}, s_i \models \phi$
11. $\mathcal{M}, s \models EG \phi \Leftrightarrow \text{there exists a path } \pi \text{ with } \pi_1 = s \text{ and for all } s_i \text{ along the path, we have } \mathcal{M}, s_i \models \phi$
12. $\mathcal{M}, s \models AF \phi \Leftrightarrow \text{for all paths } \pi \text{ with } \pi_1 = s \text{ there exists an } s_i \text{ such that } \mathcal{M}, s_i \models \phi$
13. $\mathcal{M}, s \models EF \phi \Leftrightarrow \text{there exists a path } \pi \text{ with } \pi_1 = s \text{ for which there exists an } s_i \text{ such that } \mathcal{M}, s_i \models \phi$
14. $\mathcal{M}, s \models A[\phi_1 U \phi_2] \Leftrightarrow \text{for all paths } \pi \text{ with } \pi_1 = s \text{ there exists an } s_i \text{ such that } \mathcal{M}, s_i \models \phi_2 \text{ and for each } j < i \text{ we have } \mathcal{M}, s_j \models \phi_1$
15. $\mathcal{M}, s \models E[\phi_1 U \phi_2] \Leftrightarrow \text{there exists a path } \pi \text{ with } \pi_1 = s \text{ for which there exists an } s_i \text{ such that } \mathcal{M}, s_i \models \phi_2 \text{ and for each } j < i \text{ we have } \mathcal{M}, s_j \models \phi_1$

It is often convenient to use $[Act]$ and $\langle Act \rangle$ for a set of action labels $Act \subseteq \mathcal{A}$ [MOSS99]:

$$[Act] \phi \stackrel{def}{=} \bigwedge_{a \in Act} [a] \phi \qquad \langle Act \rangle \phi \stackrel{def}{=} \bigvee_{a \in Act} \langle a \rangle \phi$$

Using this notation, the original (unparametrized) “next” operator of CTL can be written as:

$$AX \phi \stackrel{def}{=} [A] \phi \qquad EX \phi \stackrel{def}{=} \langle A \rangle \phi$$

The “weak until” operator mentioned above can be expressed by means of the “until” operator [Lar95]:

$$\begin{aligned} A[\phi_1 WU \phi_2] &\equiv \neg E[\neg\phi_2 U (\neg\phi_1 \wedge \neg\phi_2)] \\ E[\phi_1 WU \phi_2] &\equiv \neg A[\neg\phi_2 U (\neg\phi_1 \wedge \neg\phi_2)] \end{aligned}$$

3.4.2 GEAR Plugin

The GEAR plugin provides a user interface for verifying SLGs in jABC. In order to account for the various roles involved in working with jABC

(cf. Sect. 3.2) and their different skill sets, the plugin supports two different inspectors: the basic and the advanced view.

The *basic view* is visible on the bottom left of Fig. 3.7. It is intended for application experts and other users who want to check whether their modeled SLGs satisfy a given set of constraints. This set consists of a selection of constraints from the constraint library, which are applied to the current domain. This selection is usually performed by the domain expert (or a dedicated specification expert as described in [Bak+09]). The basic view provides a simple list of those active constraints which are displayed in form of natural language descriptions. While assembling an SLG, the active constraints are checked continuously⁴, and the list entries in the basic view are marked green or red depending on whether the corresponding constraint is satisfied or not, respectively. For further inspection, the application expert may select a particular constraint in the list: Consequently, each SIB in the currently displayed SLG is marked with a green or red box, depending on whether the selected constraint holds at this node. For instance, in the basic view in Fig. 3.7, the first constraint from the list is marked red (indicated by a “*”), meaning that it is not satisfied by the current SLG. The constraint description says “*Errors are always handled.*”. By clicking the constraint in the list, the source of the error can be identified directly in the SLG shown at the top of Fig. 3.7. All SIBs in this SLG are marked with a green box, except for **Extrude Java Class** (again indicated by a “*”). Unlike the others, this SIB misses an outgoing edge labeled “error” which models how errors should be handled, thus violating the constraint. In this case, the error can be corrected by simply adding the missing edge.

For domain experts and specification experts who are versed in the employed specification formalisms, the *advanced view* provides means for further diagnosis. As visible on the bottom right of Fig. 3.7, this view shows the formula underlying a constraint, in this case the formula associated with the constraint exemplified above. Furthermore, the view displays the syntax tree of the formula which may be used for further inspection: Each node in this tree corresponds to a subformula, and on selecting a particular node, again green or red boxes highlight the SIBs in the SLG based on whether they satisfy the selected subformula. This also works in the opposite direction: When selecting a particular SIB in the SLG, the single nodes in the syntax tree are marked green or red, so that the user is able to see which subformulas are satisfied by the SIB and which are not satisfied. Bakera et al. refer to this as *reverse checking* [Bak+09].

The advanced view supports the creation and modification of constraints by means of features such as syntax highlighting, on-the-fly syntax checking and auto completion. An additional formula manager dialog (not visible in Fig. 3.7) allows adding constraints to or removing them from the set of active constraints, and it is also used for providing the natural language descriptions displayed to

⁴ Alternatively, the check can be triggered manually via the “Check” button.

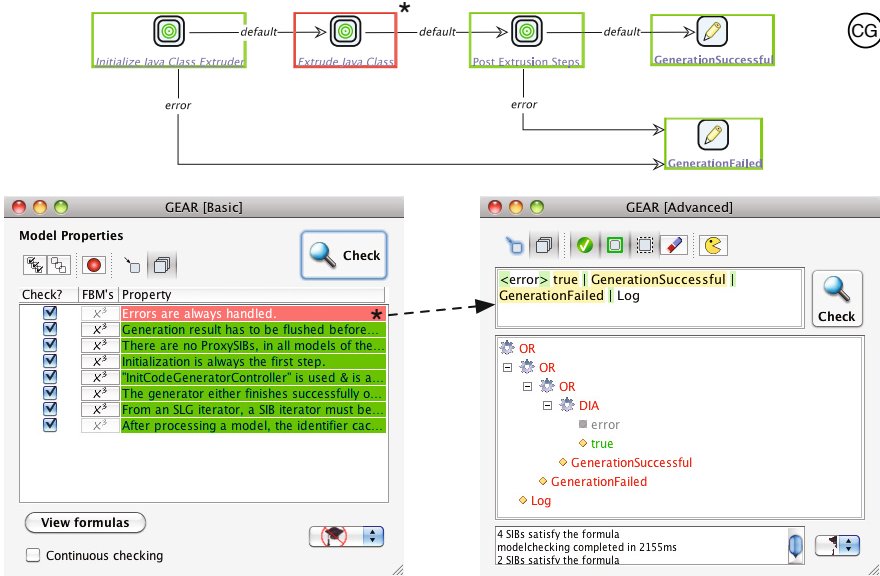


Fig. 3.7. User interface of the GEAR plugin

the application expert in the basic view. Furthermore, the advanced view allows access to GEAR's game-based facilities [Bak+09].

Finally, the GEAR plugin provides an inspector (also not shown in the figure) for manually equipping SIBs in an SLG with atomic propositions. Please note that atomic propositions may also be automatically attached to SIBs or, e.g., derived from the domain knowledge specified by the domain expert [Lam+10].

3.5 jABC as a Basis for Realizing the Genesys Approach

As mentioned at the beginning of this chapter, the demand for code generation facilities in jABC was the initial motivation for starting the Genesys project. Beyond that, jABC in general provides a suitable basis for realizing the requirements of the Genesys approach presented in Sect. 1.1. In fact, jABC and its plugins contribute to most requirements:

- *Requirement G1 - Platform Independence:* jABC's service mechanism embodied by SIBs facilitates platform independence as demanded by this requirement: SIBs represent arbitrary services (i.e., there are no restrictions whatsoever), and the corresponding implementations can be easily interchanged. The latter is performed by adding new or replacing existing service adapters, which requires no changes at the modeling level and which is thus transparent to the generator developer.

- *Requirement G2 - Reusability and Adaptability:* As pointed out in Sect. 3.1, both reusability and adaptability are rooted as very basic principles in XMDD, which are obtained by establishing libraries of models and services. The reusability of models is enabled by means of hierarchical modeling.
- *Requirement G3 - Simplicity:* As jABC essentially aims at involving non-programmers in the design and development of software systems, it is specifically designed towards simplicity.
- *Requirement G4 - Separation of Concerns:* The separation of concerns is supported by jABC via hierarchical modeling.
- *Requirement G5 - Verification and Validation:* The LocalChecker plugin and the GEAR plugin provide powerful verification mechanisms.
- *Requirement S1 - Domain-Specificity:* jABC can be tailored to a specific domain by a domain expert, using mechanisms such as SIB taxonomies, plugins and code generators.
- *Requirement S2 - Full Code Generation:* Due to the combination of models and services, jABC's SLGs represent complete and executable (sub-) systems, and thus contain all information required for the generation of complete code (cf. Sect. 2.4.4).
- *Requirement S3 - Variant Management and Product Lines:* As will be shown in Sect. 4.1.4, the mechanisms for hierarchical modeling in jABC also form an adequate technical basis for specifying variability.
- *Requirement S5 - Bootstrapping:* Bootstrapping is enabled by the executability of jABC models and by the availability of the Tracer (cf. Sect. 5.1).

In sum, jABC was an obvious choice for the reference implementation of the Genesys approach. However, the question arises as to whether other MD* approaches and tools would also be suitable for such an implementation. The remainder of this section exemplarily discusses this for several approaches related to XMDD and jABC. Please note that the presented list is not exhaustive, as the consideration of all related work⁵ of XMDD and jABC is beyond the scope of this book. Instead the following discussion is intended to provide an intuition for which general characteristics and features are necessarily required for realizing the Genesys approach.

MDA:

As pointed out in Sect. 2.3.3, platform independence and reusability (Genesys requirements *G1* and *G2*) are primary concerns of MDA. Furthermore, UML's profile mechanism (see Sect. 8.1 for details) is designed to support domain-specificity (requirement *S1*), and there are also various approaches to the verification (requirement *G5*) of UML models (e.g., [LMM99; Sch+04;

⁵ See, e.g., [Ste+07] or [Nag09] for more discussions on related work of XMDD and jABC.

JS04]). However, especially when using UML, MDA is not a suitable basis for the Genesys approach due to several reasons.

First, UML does not per se include concepts for modeling with services. There have been several proposals for extending it correspondingly (e.g., [HLT03;LS+08]), but those extensions usually add bare modeling constructs and terminology only. In particular, this means that at modeling time, there is usually no direct connection between the services reflected as corresponding constructs in the models, and their actual implementation(s). In jABC, this relationship is established by means of the service adapters attached to each SIB.

The second problem of UML is the lack of model executability. In contrast to this, jABC models are executable by design, and the Tracer provides a corresponding interpreter as well as a debugging environment. The lack of comparable facilities in UML particularly impedes the realization of the Genesys requirements *S2 - Full Code Generation* and *S5 - Bootstrapping*. Sect. 2.4.4 already pointed out that initiatives like Executable UML aim at adding executability to UML, but typically come at the expense of the abstraction provided by the models.

Finally, this lack of abstraction, which is often criticized for UML in general (cf. Sect. 2.3.3), also impinges upon simplicity (requirement *G3*).

BPM:

As mentioned in Sect. 2.6, approaches in the realm of BPM very often employ graphical modeling on the basis of services. In consequence, executability is a natural characteristic of the models occurring in those approaches. Sect. 2.3.6 listed several examples of process engines that execute the various notations available in BPM. Just like for UML, there are also several approaches to verification that apply to different process notations used in BPM [Mor08].

However, the Genesys approach typically requires executability for rapid prototyping, fast debugging during development as well as for realizing full code generation and bootstrapping. After a code generator is completed, it is typically translated into a desired implementation language running on a particular host platform (such as the JVM). In contrast to this, in BPM approaches, process engines *are* the host platform rather than primarily being a helpful device during modeling. Consequently, those engines usually provide features that aim at supporting the operation of large software systems, such as scalability, persistence or long-running transactions. While useful for the typical application scenarios of BPM, those features are not required for the development of code generators and thus provide a significant overhead. The lightweight execution enabled by jABC's Tracer is much more suitable in this context.

Another fundamental difference can be observed for domain-specificity. Due to the fact that jABC can be tailored to specific domains by design (as described in Sect. 3.2), it can in particular be easily customized for the domain "code generation". However, tools for BPM usually are optimized for and thus restricted to one specific domain – business processes. Consequently,

BPM tools are not intended at all for the development of code generators, and thus are by their very nature not able to support a generator developer like a customized domain-specific tool.

DSM and Language Workbenches:

Sect. 2.3.4 and 2.3.5 pointed out that the DSM approach and the class of tools called language workbenches include domain-specificity by definition: Both are based on providing dedicated development environments for the creation of domain-specific solutions. Due to this very general orientation, all requirements of the Genesys approach could be realized with DSM or language workbenches (excluding those language workbenches that only focus on textual DSLs, such as Xtext or Spoofax).

Realizing the Genesys approach this way would include the design of a graphical DSL that allows modeling code generators on the basis of services, and, where required, the implementation of corresponding specific tooling (e.g., an interpreter for execution, and tools for verification and testing). Consequently, such an approach would have cost significantly more development time than required for the jABC-based solution presented in this book. As jABC already provides a customizable modeling language, a corresponding modeling environment, useful plugins and several libraries of ready-made services (e.g., the Common SIBs), it was clearly preferable to DSM and language workbenches.

The Genesys Framework and Case Studies

The Genesys Framework

The Genesys code generation framework is a reference implementation of the ideas that constitute the Genesys approach presented in this book. As mentioned in Sect. 3, the jABC framework and its underlying XMDD approach form the technical and conceptual basis for this implementation. At the same time, jABC is also an appropriate domain for applying the Genesys framework in case studies (see Chap. 5).

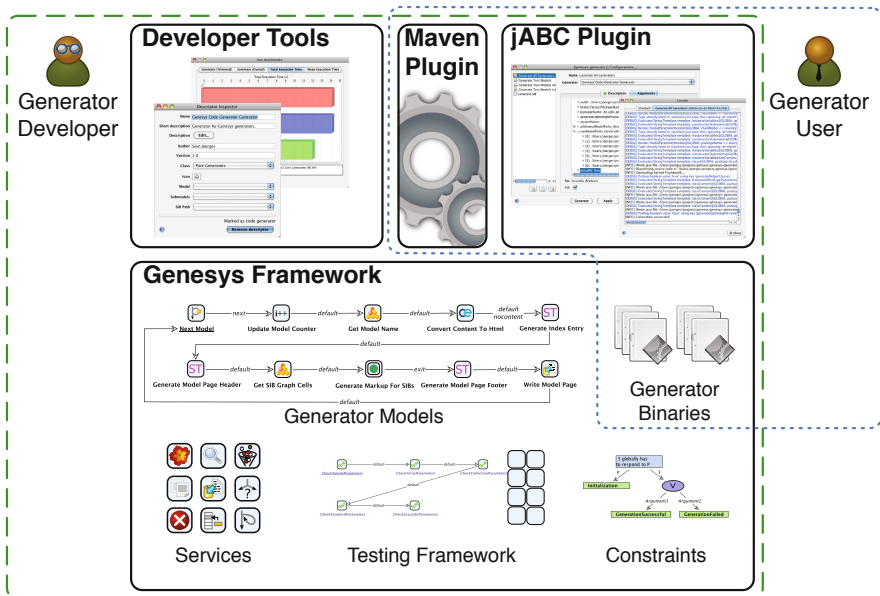


Fig. 4.1. Genesys architecture and involved roles

Fig. 4.1 shows how the reference implementation is organized, the central part being the actual *framework*, which consists of the following components:

Services: The framework provides a library of services that cover typical functionality required for most code generators, such as type conversion, identifier generation, model transformations and code beautification (*Requirement G2 - Reusability and Adaptability*). These services are available as SIBs (cf. Sect. 3.2), so that they can be used as atomic building blocks for code generator models built with jABC. Sect. 4.1 further elaborates on this service library.

Generator Models: As described in Sect. 3.2, XMDD does not only support the reuse of services, but also of entire models (*Requirement G2 - Reusability and Adaptability*). Consequently, the framework contains a library of code generator models which realize further typical functionality such as loading and traversing input models, e.g., jABC's SLGs or EMF models (cf. Chap. 7). Just like the atomic services mentioned above, these models can be directly reused as macros when building a new code generator (thus representing ready-made *features* in the sense of XMDD, see Sect. 3.1). They can also serve as patterns which are instantiated or adapted for new code generators. Furthermore, the model library contains most code generators created in the case studies which will be presented in Chap. 5–8. The rationale behind this is that each new code generator contributes to this library of models, so that the available repertoire and the potential for reuse is growing continuously (*Requirement G2 - Reusability and Adaptability*).

Generator Binaries: In order to be accessible by tools and users, the framework also includes all code generators as compiled Java classes. For this purpose, each modeled code generator is translated to an appropriate Java class via the *Genesys Code Generator Generator* (see Sect. 5.2.6).

Testing Framework: As an addition to manually testing a code generator by executing its models with the Tracer (cf. Sect. 3.3), the Genesys framework also includes an approach for the automated model-driven testing of code generators. In this approach, test cases as well as the corresponding test inputs are also modeled as SLGs. Subsequently, dedicated code generators (again developed with Genesys) translate these SLGs into test scripts or test programs running on a desired test platform. Sect. 6.3 elaborates on the details of this approach, which is currently realized for jABC code generators, i.e., those which support SLGs as input models (see Chap. 5). Accordingly, the provided facilities include a library of SIBs for building test cases and test data models, a collection of standard test cases covering the domain of jABC models as well as a testing strategy for checking whether the execution semantics of a model is retained by the code generation (cf. Sect. 6.3.1).