# 2 Use of Models in Software Engineering

In this chapter we discuss the use of models in software engineering. We examine the relation of MTCC to the fields of model-driven software development and model-driven testing.

Based on the definition of Stachowiak [Sta73] of models as abstractions with a purpose and a representation, the use of models is a well established practice in software engineering as well as in computer science as a whole. Models are not only used in approaches that are explicitly model-driven but are employed in a number of different forms and for a multitude of tasks. We consider an approach model-driven if models are the primary artifacts in a process from which the different implementation artifacts are generated. Models are used to abstract data and control flow in programming languages [Sco05]. The design of a suitable domain model [Fow03] is a central part in the development of complex software systems and the elicitation of a suitable fault model [Bin99] is a central part of the testing process. Section 2.1 discusses the definition of models we employ for MTCC and examines the notions of meta models and model transformations.

Current model-driven processes and technologies are characterized by the utilization of formal models on a level of abstraction above the specific implementation platform and by the use of model transformation, generally for the purpose of code generation, in order to bridge the implementation gap [FR07] between an abstract model and a specific platform.

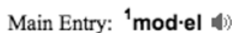
A number of different approaches for model-driven development exist that influence the MTCC approach to different degrees. MTCC is closest to the generative programming approach regarding the used models and the high degree to which the details of the implementation domain are abstracted. We introduce these approaches in Section 2.2 and compare them with MTCC.

One concrete use of models is model-driven testing, the verification of a software system based on a model of the system. Depending on the respective approach, the tested aspects of a system and its operating environment are represented by models that support the automatic generation of test cases and the allow an automated assessment of the success of the each generated tests. We discuss model-driven testing in Section 2.3 and relate it to MTCC.

## 2.1 Roles and Properties of Models

The term *model* has a number of different meanings, depending on the context in which its is used. Figure 2.1 gives an overview of 14 different definitions for the term model taken from the online version of the Merriam-Webster Dictionary[1].

For software engineering, the definitions of model as *structural design* and *a system of postulates, data, and inferences presented as a mathematical description of an entity or state of affairs; also : a computer simulation based on such a system* are of particular relevance as they allude to the prescriptive and descriptive aspects of modeling [Béz05].



Main Entry: [1]**mod·el**
Pronunciation: \ˈmä-dᵊl\
Function: *noun*
Etymology: Middle French *modelle,* from Old Italian *modello,* from Vulgar Latin **modellus,* from Latin *modulus* small measure, from *modus*
Date: 1575

**1** *obsolete* **:** a set of plans for a building
**2** *dialect British* **:** COPY, IMAGE
**3 :** structural design <a home on the *model* of an old farmhouse>
**4 :** a usually miniature representation of something; *also* **:** a pattern of something to be made
**5 :** an example for imitation or emulation
**6 :** a person or thing that serves as a pattern for an artist; *especially* **:** one who poses for an artist
**7 :** ARCHETYPE
**8 :** an organism whose appearance a mimic imitates
**9 :** one who is employed to display clothes or other merchandise
**10 a :** a type or design of clothing **b :** a type or design of product (as a car)
**11 :** a description or analogy used to help visualize something (as an atom) that cannot be directly observed
**12 :** a system of postulates, data, and inferences presented as a mathematical description of an entity or state of affairs; *also* **:** a computer simulation based on such a system <climate *models*>
**13 :** VERSION 3
**14 :** ANIMAL MODEL

Figure 2.1: Definition of *model* in the Merriam-Webster Dictionary

[1] http://www.merriam-webster.com/dictionary/model

### 2.1.1 Attributes of Models

Greenfield [GS04] defines a model in the context of software development as follows: *"... a model is an abstract description of software that hides information about some aspects of the software to present a simpler description of others..."*. Evans [Eva04] gives the following definition: *"A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain"* where domain is defined as follows: *"A sphere of knowledge, influence, or activity."*. Stahl [SV06] define the meaning of a model such: *"A model is a abstract representation of structure, function or behavior of a system."*

According to Stachowiak [Sta73], a model is characterized by three properties (translation by the author):

- **Representation property** *Models are reproductions or delineation of natural or artificial objects*
- **Abstraction property** *Models do not capture all attributes of represented original but only those that appear relevant to the creators and/or users of the model*
- **Pragmatic property** *Models are not always unambiguously assigned to their original. They fulfill a substitution role a) for certain subjects, b) for certain periods of time and c) under constraints on certain mental or physical operations*

### 2.1.2 Types of Models

A common aspect of all definitions is the concept of abstraction. For this work we define models as abstractions with a purpose [Lud03]. In the following, we further differentiate models by two characteristics:

- Models may be formal or informal. We consider a model formal when it is represented in a way that allows the automatic processing or transformation of the model for its intended purpose. The structure and semantics of a formal model are defined by its meta model.

  Modeling languages like the UML can be used both formally and informally. When used formally in the context of the MDA [Bro04], the MOF [ALP05] serves as a meta model for the UML. Besides its use for formal modeling, the UML can also be used as a sketch in order to outline the structure of systems.

Whether a modeling language is formal or informal is independent from the question whether it uses a graphical or textual representation [HR04].

- Models can be used either prescriptively or descriptively. Prescriptive models are used for the design of systems that are yet to be constructed, descriptive models represent an system that already exists [Béz05].

From the definition of a model as an abstraction with a purpose follows that the subject represented by the model and the goal of modeling process must be considered in every modeling situation. The subject and the purpose of a model define the domain that is covered by the model. This domain determines the criteria that stipulate which aspects of reality have to be representable by the model at what level of detail.

#### 2.1.2.1 Meta Model

A meta model defines the structure of a model as well as its semantics. The semantics that can be expressed are specific for a domain, therefore, the meta model is a representation of that domain. Figure 2.2 examines the relationship between formal models, their domain and the various concepts that define a meta model. Models are expressed in a domain specific language (DSL). A DSL defines a concrete syntax or formal notation used to express concrete models. The abstract syntax defines the basis concepts available for model construction, it is determined by static semantics of the meta model.

Notwithstanding the existence of a multitude of standards and approaches for meta modeling [KK02, BSM$^{+}$03, ALP05], meta models and their use are an active field of research. This is also true for the integration of different meta models [ES06, BFFR06] and the representation of constraints on meta models [CT06]. We argue that neither research nor practice have yet succeeded in establishing an universal meta modeling standard.

Meta models are related to ontologies [SK06]. While ontologies aim at the presentation of knowledge, the purpose of meta models is the definition of a formal language.

Figure 2.2 illustrates one important property of models: The separation of the representation of a model, its concrete syntax, from its semantics. This property allows the use of multiple representations for one meta model, for instance the textual representation of UML [Spi03] or — in the case of MTCC — the representation of the feature model through a graphical editor.

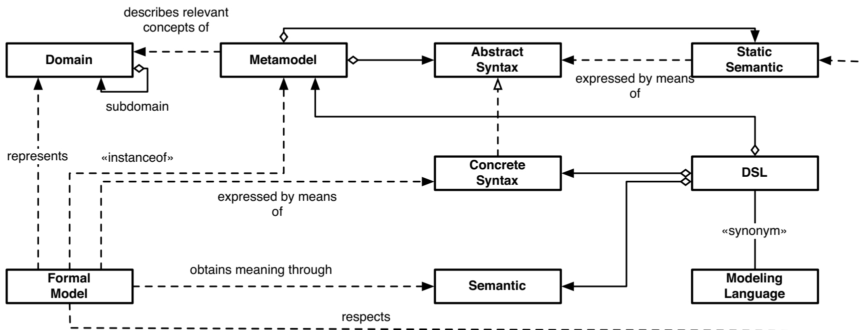



Figure 2.2: Concepts of model-driven software development [SV06]

## 2.2 Model-Driven Software Development

The purpose of model-driven software development is to close the gap between the problem domain and the solution domain [SV06]. Models abstract from the technical details of an implementation and facilitate the use of concepts from the subject domain not only in requirements engineering and design, but also during the implementation of a system.

France [FR07] express this goal as follows: *"current research in the area of model driven engineering (MDE) is primarily concerned with reducing the gap between problem and software implementation domains through the use of technologies that support systematic transformation of problem-level abstractions to software implementations."*

The stated goals and claims associated with model-driven software development, in particular the MDA approach, have been the subject of critique [McN03, Tho04]. We argue that potential deficits of specific approaches do not challenge the effectiveness of model-driven software development in general but rather that the methods and tools used for any software project must consider the individual requirements of each project.

In the following, we introduce multiple approaches to model-driven software development and relate them to the MTCC. We discuss the transformation of models into implementations. Furthermore we explain the relationship of model-driven development and programming languages with a particular focus on domain specific languages [vDKV00]. We present a summary of relevant approaches to model-driven development, a more detailed comparison can be found in [Völ06a].

### 2.2.1 Model Driven Architecture

The Model Driven Architecture (MDA) is a standard for model-driven development propagated by the Object Management Group [MSU04, Uhl06].

One defining aspect of the MDA is the successive refinement of models [SV06]. A Platform Independent Model (PIM) represents functional requirements in a formal notation. The PIM is used as the basis for one or more Platform Specific Models (PSM). Each PSM is an abstract representation specific for one aspect of the implementation, for instance, the database layer or the deployment architecture. In order to create an implementation, each PSM undergoes one or more transformation steps, turning the PSM into successively more detailed models and finally implementation artifacts. The MDA emphasizes model-to-model transformations in the software construction process, as the description above illustrates.

We argue that the MTCC approach does not benefit from the advantages offered by the MDA approach because the increase in complexity that the adoption of the MDA would cause is not worthwhile.

### 2.2.2 Architecture-Centric Model-Driven Software Development

Architecture-Centric model-driven Software Development (AC-MDSD) is an approach introduced by Stahl and Voelter [SV06]. The purpose of AC-MDSD is the *"holistic automation during the development of infrastructure code respectively the minimization of redundant, technical code in software development"*. A concept central to the approach is the platform, the total of all parts of the infrastructure that are used for the implementation. Compared to the MDA, the focus of AC-MDSD is more on the abstraction of technical detail than on the complete representation and implementation of a system with models. We argue that this approach is more pragmatic than the vision offered by the MDA. MTCC shares the goal of AC-MDSD to abstract from infrastructure code; in the case of MTCC, test code and code used to interface with the testee.

### 2.2.3 Generative Programming

With Generative Programming, Czarnecki [CE00] introduced an approach that aims at creating complete implementations of software systems, optimized for specific requirements, based on a library of implementation artifacts and a formal representation of available configurations. Czarnecki describes the approach as follows: *"Generative software development is a system-family approach, which*

*focuses on automating the creation of system-family members: a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific languages"* [Cza05]. Like Generative Programming, MTCC considers families of systems, more specific families of tests.

### 2.2.4 Generation of Implementations

In order to facilitate the creation of implementation artifacts from models, a transformation step is necessary [CLW06]. With the execution of one or more transformation steps, implementation details and models are integrated and successively more detailed models are created. A transformation step can either be a Model2Code transformation or a Model2Model transformation [SV06]. While a Model2Model transformation generally starts with an input model and generates another more detailed, model, Model2Code transformations generate an implementation artifact, usually program code. A number of different model transformation approaches exist that address different goals [CH03] and vary in the complexity and suitability for complex transformation scenarios. The test cases generated in the MTCC exhibit only minimal complexity. Accordingly, we limit our consideration of possible transformation strategies to template-based [Her03, Voe03] code generation.

### 2.2.5 Models and Programming Languages

Model-driven development shares a significant number of concepts with programming languages and compiler technology. Considerable parallels exist between model transformation and compilation — both processes create implementations from abstract representations. The concepts used by programming languages to describe both data and control flow [Sco05] and the rules that govern their use and possible combinations in a program correspond to a meta model.

Domain specific languages [vDKV00, TK04] exist for a wide variety of application domains [MHS05, Hud96]. Such languages have the same purpose as models in MTCC expressing the concepts of a specific application domain — they implement problem-specific abstractions [GKR+06]. Implementation techniques for DSLs are similar to those used in model-driven development [Völ07, Völ06c], especially for approaches that use code generation.

Another approach that serves to illustrate the close relation between model-driven development and programming languages is Languages Oriented Programming. Fowler gives an overview of one such approach [Fow05c, Fow05b, Fow05a], the MPS environment [Dmi04]. MPS aims to facilitate the inte-

grated development of a meta model, an editor for modeling, and transformations for implementing DSLs. The intentional software approach is similar in its goals [Sim95, SCC06].

MTCC uses a graphical editor to present models to domain experts. The XML-based representation of MTCC models can be considered a domain specific language for system and test description.

## 2.3 Model-Driven Testing

Testing is always Model-Driven in the sense that it is based on a fault model. A fault model considers a system with regards to its likely faults. Binder gives the following definition of a fault model*"(A Fault Model) identifies relationships and components of the system under test that are most likely to have faults. It may be based on common sense, experience, suspicion, analysis, or experiment..."* [Bin99].

Similar to models used in the construction of systems, a fault model does not have to be formal to be useful for testing, it may take the form of a list containing the functionality of a system to be tested. Formal models that are suitable as a basis for automatic testing are called testable [Bin99]. The use of testable models for the automatic verification of software is the subject of model-driven testing.

Figure 2.3 displays an example of a model-driven testing process and the concepts that are relevant during the phases of the process.

- In a first step, a model that can be utilized for the automatic generation of tests is realized based on the requirements of the examined system. The model represents those aspects of the behavior and state of the testee that are relevant for testing. For any possible input to the system, it provides the outputs expected from the system. The models serve as an oracle.

- In the Test Case Specification phase, criteria for the selection of tests are defined and formalized. Criteria include the test coverage by different metrics as well as random selection of tests and the use of previously recorded user interactions with the system. The Test Case Specification is a formalization of this criteria that facilitates the automatic selection of tests cases.

- Tests are generated and executed on the SUT. The adaptor in Figure 2.3 corresponds to the concept of the test runner in MTCC. The results of all executed tests are recorded.

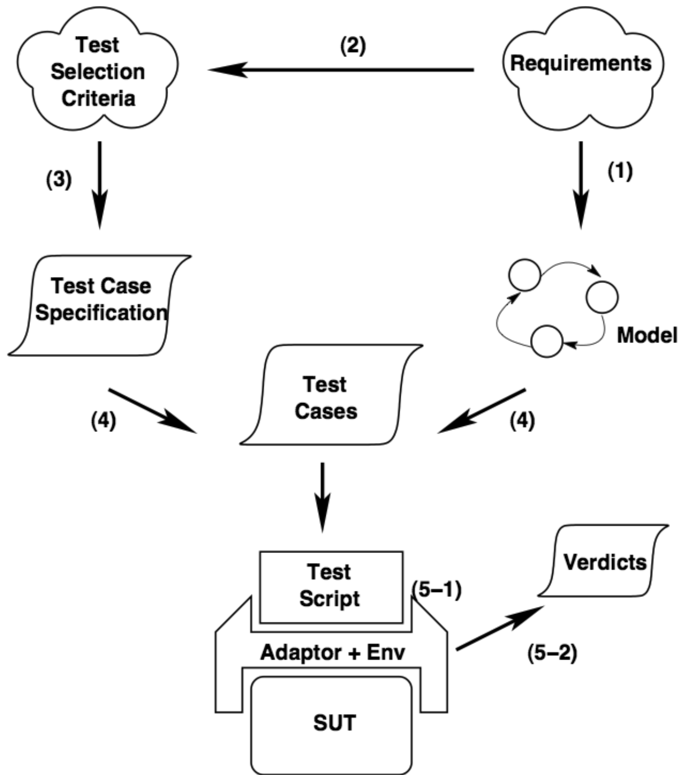



Figure 2.3: A process for model-driven testing [UPL05]

### 2.3.1 Types of Model-Driven Testing

As in model-driven development, many approaches to model-driven testing exist. A criterion by which different approaches can be differentiated from MTCC is *Redundancy* [UPL05]. Redundancy differentiates between approaches that reuse models created for the construction of a system from those that use specialized models created only for testing. Bertolino uses the term test-based modeling [Ber07] for the latter group.

Approaches that aim at reusing or extending models or modeling tools created for the construction of software are limited by the modeling languages used in

this process, often the UML. As a consequence, there is a great number of approaches to model-driven testing that are based on the UML [CCD04, BLC05, SS08, LJX+04, KWD05, HVR04].

One application of the UML for testing is the UML2 Testing profile [SDGR03, BDG+03, Dai04]. The purpose of the U2TP is to provide an UML profile that adds test-relevant concepts to the UML2 core language and thus allows the modeling of tests using UML specific tooling. The U2TP can be regarded as model-driven development applied to the testing domain. By envisioning intellectual test specification, the U2TP bears similarities to MTCC, but differs in the modeling language and the support for modeling by domain experts.

Test-based modeling approaches are not limited to any particular modeling language. Utting [UPL05] present an overview over approaches to model-driven testing and the models used in these approaches. Finite State Machines [EFW01, AOA05], decision trees and decision tables are frequently used as models for testing purposes [Bin99].

### 2.3.2 Relation to MTCC

Figure 2.4 displays a taxonomy of model-driven testing that classifies approaches to model-driven testing along seven dimensions. Not all dimensions defined by the taxonomy can be applied to MTCC, in the following we only discuss the relevant classification criteria.

**Subject** MTCC models represent a testee from a system family of testees. The test-relevant Services of a testee as well as the Test Steps that exercise and verify these Services are modeled.

**Redundancy** The models constructed for the MTCC are only used for testing.

**On/Offline** MTCC generates and executes test cases offline.

**Test Selection Criteria** MTCC envision test case construction by domain experts. Domain experts select and implement those test that they consider relevant based on their knowledge of the requirements of the system.

## 2.4 Chapter Summary

A number of different meanings exist for the term *model*. In this work, a model is a formal abstraction that serves a specific purpose. A model is considered formal if its structure and semantics conform to an implicit or explicit meta-model.

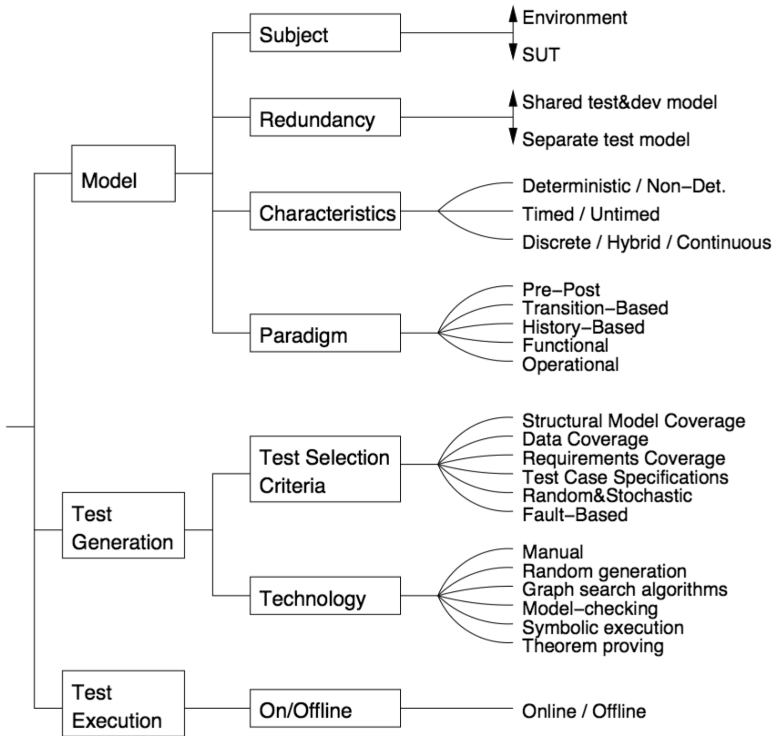



Figure 2.4: A taxonomy of model-driven testing [UPL05]

Model-Driven Software Development attempts to solve the *Problem Implementation Gap* by using formal models that represent concept of the application domain. Applications are modeled and the resulting models are used, optionally after further refinements, to generate implementation artifacts. MTCC is a model-driven approach that uses models to represent SUTs and tests that exercise a SUT.

Model-Driven Testing uses formal models of a SUT and optionally the environment of the SUT to generate tests and to assess the success or failure of these tests. MTCC is closely related to model-driven testing approaches but differs in that tests are not generated but constructed by domain experts.