From wind-tunnels to Navier-Stokes equations to circuit diagrams to finite-element models of buildings, engineers in all fields of engineering construct and analyze models. Fundamentally, modeling addresses two problems in engineering. First, analysis and test cannot wait until the actual artifact is constructed, whether that artifact is a building or a software system. Second, it is impractical to test the actual artifact as thoroughly as we wish, whether that means subjecting it to all foreseeable hurricane and earthquake forces, or to all possible program states and inputs. Models permit us to start analysis earlier and repeat it as a design evolves, and allows us to apply analytic methods that cover a much larger class of scenarios than we can explicitly test. Importantly, many of these analyses may be automated.

This chapter presents some basic concepts in models of software and some families of models that are used in a wide variety of testing and analysis techniques. Several of the analysis and testing techniques described in subsequent chapters use and specialize these basic models. The fundamental concepts and trade-offs in the design of models is necessary for a full understanding of those test and analysis techniques, and is a foundation for devising new techniques and models to solve domain-specific problems.

5.1 Overview

A model is a representation that is simpler than the artifact it represents but preserves (or at least approximates) some important attributes of the actual artifact. Our concern in this chapter is with models of program execution, and not with models of other (equally important) attributes such as the effort required to develop the software or its usability. A good model of (or, more precisely, a good class of models) must typically be:

Compact A model must be representable and manipulable in a reasonably compact form. What is "reasonably compact" depends largely on how the model will be used. Models intended for human inspection and reasoning must be small enough to be comprehensible. Models intended solely for automated analysis may be far too large and complex for human comprehension, but must still be sufficiently small or regular for computer processing.

Predictive A model used in analysis or design must represent some salient characteristics of the modeled artifact well enough to distinguish between "good" and "bad" outcomes of analysis, with respect to those characteristics.

Typically, no single model represents all characteristics well enough to be useful for all kinds of analysis. One does not, for example, use the same model to predict airflow over an aircraft fuselage and to design internal layout for efficient passenger loading and safe emergency exit.

Semantically meaningful Beyond distinguishing between predictions of success and failure, it is usually necessary to interpret analysis results in a way that permits diagnosis of the causes of failure. If a finite-element model of a building predicts collapse in a category five hurricane, we want to know enough about that collapse

to suggest revisions to the design. Likewise, if a model of an accounting system predicts a failure when used concurrently by several clients, we need a description of that failure sufficient to suggest possible revisions.

Sufficiently general Models intended for analysis of some important characteristic (e.g., withstanding earthquakes or concurrent operation by many clients) must be general enough for practical use in the intended domain of application.

We may sometimes tolerate limits on design imposed by limitations of our modeling and analysis techniques. For example, we may choose a conventional bridge design over a novel design because we have confidence in analysis techniques for the former but not the latter, and we may choose conventional concurrency control protocols over novel approaches for the same reason. However, if a program analysis technique for C programs is applicable only to programs without pointer variables, we are unlikely to find much use for it.

Since design models are intended partly to aid in making and evaluating design decisions, they should share these characteristics with models constructed primarily for analysis. However, some kinds of models - notably the widely used UML design notations - are designed primarily for human communication, with less attention to semantic meaning and prediction.

Models are often used indirectly in evaluating an artifact. For example, some models are not themselves analyzed, but are used to guide test case selection. In such cases, the qualities of being predictive and semantically meaningful apply to the model together with the analysis or testing technique applied to another artifact, typically the actual program or system. Graph Representations We often use directed graphs to represent models of programs. Usually we draw them as "box and arrow" diagrams, but to reason about them it is important to understand that they have a well-defined mathematical meaning, which we review here.

A directed graph is composed of a set of nodes N and a relation E on the set (that is, a set of ordered pairs), called the edges. It is conventional to draw the nodes as points or shapes and to draw the edges as arrows. For example: Image from book

Typically, the nodes represent entities of some kind, such as procedures or classes or regions of source code. The edges represent some relation among the entities. For example, if we represent program control flow using a directed graph model, an edge (a,b) would be interpreted as the statement "program region a can be directly followed by program region b in program execution."

We can label nodes with the names or descriptions of the entities they represent. If nodes a and b represent program regions containing assignment statements, we might draw the two nodes and an edge (a,b) connecting them in this way: Image from book

Sometimes we draw a single diagram to represent more than one directed graph, drawing the shared nodes only once. For example, we might draw a single diagram in which we express both that class B extends (is a subclass of) class A and that class B has a field that is an object of type C. We can do this by drawing edges in the "extends" relation differently than edges in the "includes" relation. Image from book

Drawings of graphs can be refined in many ways, for example, depicting some relations as attributes rather than directed edges. Important as these presentation choices may be for clear communication, only the underlying sets and relations matter for reasoning about models.

5.1.1 Modeling maturity level

It is also useful to consider the modeling maturity level of a company that is adopting model-based testing. The UML/OCL and MDA (model-driven architecture) community has identified six levels of modeling maturity for UML development models and MDA [WK03]:

- Level 0, No Specification: The software specifications are only in the heads of the developers.

- Level 1, Textual: The software specifications are written down in informal natural-language documents.

- Level 2, Text with Diagrams: The textual specifications are augmented with some high-level diagrams.

- Level 3, Models with Text: A set of models (diagrams or text with well- defined meanings) form the backbone of the specification. Natural language is used to motivate and explain the models and to fill in many details within the models. The transition from models to code is still manual, and it can be difficult to keep models up to date after changes to the code.

- Level 4, Precise Models: This is the level where MDA becomes possible, with code being generated from the model and then modified to fulfill special requirements. The model has a precise meaning, which does not rely on natural language even though natural language is still used to explain the background of the model.

- Level 5, Models Only: At this level, the model is used like a high-level programming language, the model-to-code generation is automatic and used just like a compiler, and the generated code is used directly with- out changes. In 2003, the authors commented that this level has not yet been reached anywhere in the world, but it is a good ultimate go

5.2 How to model your system

The first and most important step in modeling a system for testing is deciding on a good level of abstraction, that is, deciding which aspects of the system to include in your model and which aspects to omit. Since the model is just for verification or for test generation purposes, it does not have to specify all the behavior of the system. In the followinfg we assume that the model is used for testing. The same applies for verification. Several smaller partial models are often more useful than one huge and complex model.

For example, it may be useful to write a model for each subsystem or component and to test them independently, before writing a top-level model for the whole system. So your decisions about which operations to include in the model should be driven by your top-level test objectives. Once you have decided which aspects of the SUT you wish to model, the next step in modeling a system is to think about the data that it manages, the operations that it performs, and the subsystems that it communicates with. A good notation for this is a UML class diagram, perhaps enhanced with a few textual UML use cases for the most important operations. If you already have a UML class diagram that describes the design of the SUT, you may be able to use that as a starting point for the testing model. However, a class diagram for testing purposes should be much simpler than the full class diagram that is used for design purposes. The following are some typical simplifications:

- Focus primarily on the SUT

- Show only those classes (or subsystems) associated with the SUT and whose values will be needed in the test data

- Include only those operations that you wish to test

- Include only the data fields that are useful for modeling the behavior of the operations that will be tested

- Replace a complex data field, or a class, by a simple enumeration. This allows you to limit the test data to several carefully chosen example values (one for each value of the enumeration).

For each operation that you decide to model, you should also apply the abstraction principle to its input and output parameters. If the value of an input parameter changes the behavior of an operation, and you want to test those different behaviors, then put that input parameter into your model. Otherwise, it is generally better to leave the input parameter out of the model to keep it simple—an appropriate input value can be chosen after test generation when the abstract tests are being translated into executable tests. The difficulty of test generation is usually highly dependent on the number and range of the input parameters (in addition to the state variables), so reducing this helps to control the test generation effort. Output parameters should be modeled only if their value is useful as an oracle for the test.

> **Key Point** Design your model to meet your test objectives. When in doubt, leave it out!

Note that the operations in your model do not have to be exactly the same as the operations of the SUT. If your system has a complex operation Op, you may want to split its behavior into several cases and define one model operation Op_i for each case. On the other hand, you may want to define one model operation that corresponds to a sequence of operations in the actual system, perhaps to summarize a complex initialization sequence into a single model operation, or to reach a particular state that you want to test thoroughly.

> Key Point You can have a many-to-many relationship among the operations of your model and the operations of the SUT.

The next step is to decide which notation to use for your model. This decision is often influenced by the model-based testing tools you have available and the notations they support. But in addition to this factor, it is important to consider which style of notation is most suitable for your system. In the next section, we give an overview of the different modeling notations that are available and some guidelines for choosing an appropriate notation. After you have chosen a notation and written a model of your system in that notation, the next step is to ensure that your model is accurate. You will want to validate your model (check that it does indeed specify the behavior that you want to test) and verify it (check that it is correctly typed and consistent). This is where good tool support can help. Most tool suites offer an animation tool for simulating the behavior of your model, which helps you to validate your model. They also provide tools for checking the syntax and types in your model and may offer more sophisticated tools for checking deeper properties of your model, such as an automatic prover that tries to prove that each operation of a B machine preserves the invariant or a model checker that searches for states where no operations are enabled (deadlocked states). The final step is to use your model to generate tests. This is the subject of the next few chapters. Note that your model will continue to be validated throughout the test generation process. After you generate tests from your model and execute those tests on your system, each test that fails will point either to an error in the implementation of your system or to a mistake or inadequacy in your model. The value of model-based testing comes from the automated cross-checking between these two independent works of art: the model and the system implementation.

5.2.1 Notations for Modeling

Dozens, perhaps even hundreds, of different modeling notations have been used for modeling the functional behavior of systems.

We group them into the following paradigms, adapted from van Lamsweerde [vL00]. Pre/post (or state-based) notations: These model a system as a collection of variables, which represent a snapshot of the internal state of the sys- tem, plus some operations that modify those variables. This is similar to an object in Java or C++. Rather than the operations being defined with programming language code, each operation is usually defined by a precondition and a postcondition. Examples of these notations include B [Abr96], the UML Object Constraint Language (OCL) [WK03], the Java Modeling Language (JML) [L + 06], Spec# [Res06a], VDM [Jon90, FLM + 05] and Z [ISO02, Bow06]. Note: The traditional name for these notations is "model-based." However, this is rather confusing in our con- text, where all kinds of notations are being used to define models of the SUT. So, in this book, we call them pre/post notations. Transition-based notations: These focus on describing the transitions be- tween different states of the system. Typically, they are graphical node- and-arc notations, such as FSMs, where the nodes of the FSM represent the major states of the system and the arcs represent the actions or oper- ations of the system. Textual or tabular notations are also used to specify the

transitions. In practice, transition-based notations are often made more expressive by adding data variables, hierarchies of machines, and parallelism between machines. Examples of transition-based notations include FSMs, statecharts (e.g., UML State Machines, STATEMATE statecharts, and Simulink Stateflow charts), labeled transition systems, and I/O (input/output) automata. History-based notations: These notations model a system by describing the allowable traces of its behavior over time. Various notions of time can be used (discrete or continuous, linear or branching, points or intervals, etc.), leading to many kinds of temporal logics. We also include message-sequence charts (MSC) and related for- malisms in this group. These are graphical and textual notations for specifying sequences of interactions among components. They are often used for modeling telecommunication protocols, particularly in combination with the System Description Language (SDL)[1]. MSCs were adopted into UML, where they are called sequence diagrams, which are one kind of interaction diagram. MSCs are good for visually showing in- teractions among components, but not so good at specifying the detailed behavior of each component. So, although they are sometimes used as a basis for model-based testing, our preference is to use them to describe the generated tests. That is, they are better used for visualizing the tests that result from model-based testing than for defining the model that is the input to model-based testing. Functional notations: These describe a system as a collection of mathematical functions. The functions may be first-order only, as in the case of algebraic specifications, or higher-order, as in notations like HOL (an environment for interactive theorem proving). For example, the property push;pop = skip specifies that the pop operation undoes the effect of a push operation. Algebraic specifications tend to be more abstract and more difficult to write than other notations, so they are not widely used for model-based testing (but see [Mar95] for one test generation tool based on algebraic models). Operational notations: These describe a system as a collection of exe- cutable processes, executing in parallel. They are particularly suited to describing distributed systems and communications protocols. Examples include process algebras such as CSP and CCS on the one hand and Petri net notations on the other hand. Statistical notations: These describe a system by a probabilistic model of the events and input values. They tend to be used to model environ- ments rather than SUTs. For example, Markov chains are often used to model expected usage profiles, so the generated tests exercise that usage profile. Statistical notations are good for specifying distributions of events and test inputs for the SUT but are generally weak at predicting the expected 1 .... .64 chapter 3 A Model of Your System outputs of the SUT; therefore, with only a statistical model it is not usu- ally possible to generate automated oracles as part of the tests. However, it is possible to combine a statistical model with one that models the SUT behavior. This allows the statistical model to drive the choice of test sequences and inputs, while the other model predicts the expected outputs of the SUT. Data-flow notations: These notations concentrate on the flow of data through the SUT, rather than its control flow. Some examples of this style are Lustre [MA00] and the block diagram notations that are used in Matlab Simulink[2] for

---

[1]See the SDL Forum Society, http://www.sdl-forum.org, for more details of SDL and MSC
[2]See the MathWorks website, http://www.mathworks.com, for information on the Simulink product

the modeling of continuous systems.

5.2.2 Choosing a Notation

For model-based testing, the transition-based notations and the pre/post notations are the most used for developing behavioral models of the SUT. Which notation will be the best for modeling your SUT? In addition to practical factors, such as the availability of model-based testing tools for each notation and the degree of familiarity that you have with some notations, the answer will depend on the characteristics of your SUT. The basic guideline is to look at whether your system is more data- oriented or control-oriented. A data-oriented system typically has several state variables, with rich types such as sets, relations, maps, and sequences of values. The operations of a data-oriented system operate on that data to ac- cess and manipulate it. Data-oriented systems are most easily specified using pre/post notations, like B, which offer powerful libraries of data structures. In a control-oriented system, the set of available operations varies according to which state the system is in. For example, in the drink vending machine that we will study later in this chapter, some operations are enabled only when the machine is out of service, and others are enabled only when the machine is in service. Control-oriented systems are most easily specified using transition-based notations, such as UML state machines, because the set of transitions can be different for each state machine node.

> Key Point Pre/post notations are best for data-oriented systems. Transition-based notations are best for control-oriented systems.

Of course, your system may not be so easy to classify. It may show signs of being both data-oriented and control-oriented. These classifications are not opposites, but rather two independent dimensions, so you must make a judgment call about which dimension is dominant in your system. We close by noting that it is always possible to specify control-oriented aspects of a system in a pre/post notation, and it is usually possible to specify data-oriented systems in a transition-based notation. If several "states" of your system have different behavior and could easily be modeled by nodes in a transition-system, you can still model this using a pre/post notation. You simply introduce one or more state variables to tell you which of those "nodes" your system is currently in (like state : {InService,OutOfService}), and then you add appropriate preconditions (like state = InService) or if-then-else conditions into the operations to enable the desired operations. Operations that correspond to a transition from one state to another must also update the state variables to reflect this change (like state := InService). On the other hand, a transition-based notation may have good enough support for complex data structures that it can be used for data-oriented systems. With UML state machines, OCL offers Set, OrderedSet, Bag, and Sequence data types, and associations among classes can be used to model many-to-many relations, many-to-one functions, one-to-one functions, and so on. OCL has fewer built-in operators than the B toolkit, but has a good set of quantifiers that can be used to express sophisticated queries. So choosing the "wrong" notation may

---

family.

7

make it a little more difficult to model your system. The goal is to choose the notation that suits your system best and one for which you have good tool support. Whatever notation you choose, the important thing for model-based testing is that the notation be formal. Formal means that the notation has a precise and unambiguous meaning, so that the behavior of your model can be understood and manipulated by various tools. This precise meaning makes it possible to simulate the execution of the model and to use the model as an oracle by predicting the expected output of the SUT or checking the actual output of the SUT against the model. Moreover, the precise semantics of modeling notations allow tools to perform deep consistency checks on your model, such as proving that every operation preserves the desired properties of your data structures (the invariant) or that all states are reachable.

Key Point Model-based testing requires an accurate model, written in a formal modeling notation that has precise semantics.

For example, a UML class diagram by itself does not give enough detail for test generation. One must add behavioral detail in some way, perhaps by writing OCL preconditions and postconditions for the methods of the class or perhaps by writing a state machine diagram with detailed transitions.

5.3 Declarative vs operational notations

Roughly speaking, there are two ways to describe a system: *operationally* (also called "imperative") and *declaratively.* An operational modeller asks "how would I make X happen?". A declarative modeller asks "how would I recognize that X has happened?". In many contexts, it is more natural and concise to use a declarative description. Often one need only to describe the rules of the game and what it means to have won, and not even think about what moves will be needed (and which must be avoided) in order to win.

5.4 Finite State Machines

5.5 Logic

5.6 Temporal logic

LTL /CTL - altri lucidi

5.7 Abstract State Machines

lucidi Model analysis and verification *Model simulation (scenario based)
    *Model review

5.8 formal verification

5.9 model checking