St. Pölten University of Applied Sciences

/informatik & security

/fh/// st. pölten

Modeling
Theoretical Computer Science

Dipl.-Ing. Hubert Schölnast, BSc
06. September 2021

St. Pölten University of Applied Sciences, Campus-Platz 1, 3100 St. Pölten, T: +43 (2742) 313 228, F: +43 (2742) 313 228-339, E: csc@fhstp.ac.at, I: www.fhstp.ac.at

Table of Contents

1 What is a model? 3

1.1 Spelling ("model" vs. "modell") 3
1.2 What is it good for? 3
1.3 So what is a model? 4

2 How do you make a model? 5

3 Degree of accuracy 5

3.1 Limitations 6

4 Properties of a good model 7

1 What is a model?

1.1 Spelling ("model" vs. "modell")

Before we begin: Is it *"model"* (single L) or *"modell"* (double-L)?

Both words are correct, and both words mean the same thing. And although *"model"* appears to be American English and *"modell"* is of British origin, you will find both words with exactly the same frequency in texts written in British English (figures from 2015 to 2019, before 2015 *"modell"* was more often used in UK), while in texts from the USA *"model"* is used about five times as often as *"modell"*. Thus, both versions are correct in both major versions of English, but "model" is used more frequently, so this document prefers that spelling.

1.2 What is it good for?

Maybe we can figure out what a model is if we understand what models are good for.

In software development, many people do their best to write code that meets the customer's requirements. One problem is that sometimes the programmers have a different understanding of those requirements than the customer. Another problem is that even if all programmers have the same understanding of the requirements, they need interfaces to send and receive data, and sometimes two programmers have a different understanding of the interface they both designed. And the next problem is that programmers are human beings who make mistakes.

You all know the consequences: Faulty software and errors and bugs everywhere in the code. These errors cost billions of euros and cost thousands of lives.

How can you solve this really serious problem? The answer is: test the software before you ship it.

Sound like a good idea?

Well, one problem is that sometimes the testers have a different understanding of the requirements than the programmers and the customer. Another problem is that even if the testers and the programmers have the same understanding of the requirements, the testers need test software written by someone, so you need requirements for the test tools. And the next problem is that testers are humans who make mistakes.

Is that still a good idea?

Wouldn't it be great to have a really clean and mathematically proven solution? Well, there is such a solution, and its name is "model checking". You need an exact description of the requirements in a mathematical language, and you also need an exact description of the software in that language, and then you can use mathematical tools like reasoning to decide if

the software meets all the requirements. And if you are lucky, this mathematical proof of correctness takes an acceptable amount of time.

And now let's look at the topics in our course:

- "acceptable amount of time" = computational complexity.
- "write the requirements in a mathematical language" = formal languages and automata
- "decide whether the software satisfies all the requirements" = decidability
- "logical reasoning" = logic

1.3 So what is a model?

This should be clearer now: A model is a set of specifications that together represent the requirements for something that needs to be built.

The above example comes from software development, because in practice there are tools that can be used to make software more robust. But the general idea of a technical description of requirements can be used anywhere where something new is to be built.

If you are building a house, someone draws a plan. This plan shows exactly where the walls should be, how thick they are, how high they are, and additional documents specify what material they must be made of. There is another plan that shows where the water pipes should run, and another plan for the electrical installation. And there are documents that explain in what colors the walls should be painted, what furniture there should be, and what the floor should be made of.

All these plans and documents together are a formal description of this particular house. This is the *model* of the house, in the sense in which we use that term here.

And everyone involved in building that house does it according to that model. And when the house is finished, the owner checks to see if the real house exactly meets all the specifications of that model.

The plans and design documents of a house are usually not written in a mathematical language, because there is no mathematical proof if the real house meets the specifications, but the idea is the same.

2 How do you make a model?

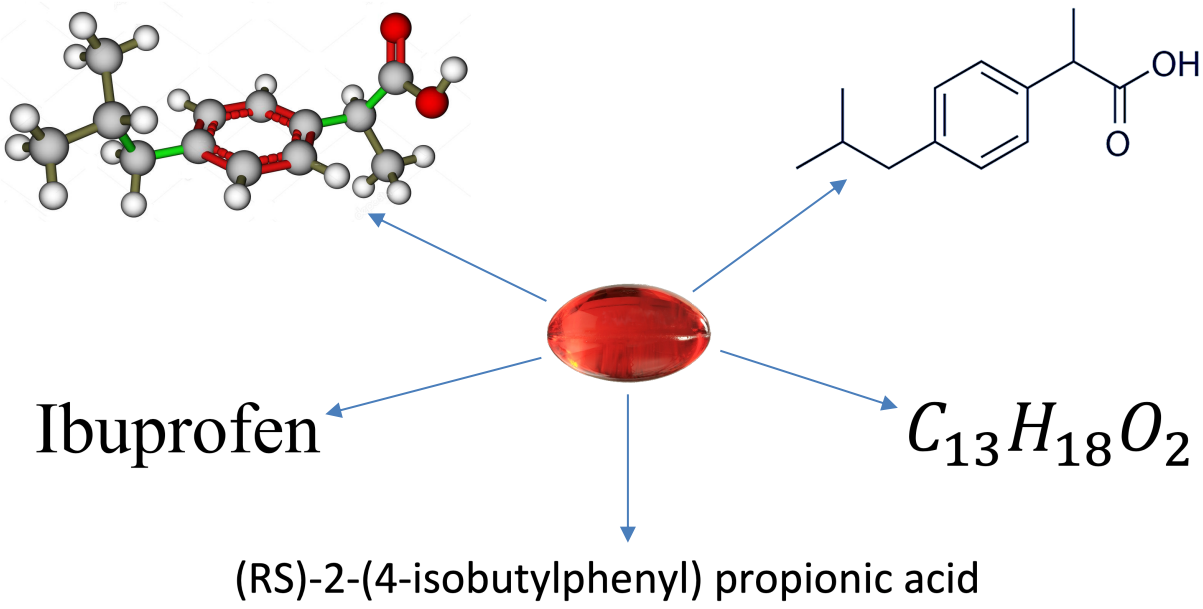
If you really want to do model checking in software development, you need to learn special modeling languages, and you need to write your code in special programming languages. But you won't be doing any model checking here in this curse.

All you have to do is find a description of your problem that is technical and formalized enough to be understood and reviewed by a person who doesn't speak any of the languages you do. So you have to find a non-verbal description. Many (if not all) real problems are realizations of mathematical problems. So you can try to find a mathematical formula that describes your problem.

If the problem your project is trying to solve is a software design problem, you can use the full set of Unified Modeling Language (UML) tools to describe use cases and workflows.

3 Degree of accuracy

Note that the design of the model also strongly depends on the purpose for which it is to be used. Below are several models of a chemical substance that represent the active ingredient of an analgesic in different ways.



Another example is the scientific model of gravity. It can be described in different degrees of accurcy:

- "Things fall down"
- Speed of falling things: $v = \sqrt{2 \cdot g \cdot h}$
  - v = velocity
  - g = gravitational acceleration; $g = 9.80665\ \frac{m}{s^2}$
  - h = height
- Newton's law of universal gravitation $F = G \cdot \frac{m_1 \cdot m_2}{r^2}$
  - F = force
  - G = gravitational constant; $G = 6.67430 \frac{m^3}{kg \cdot s^2}$
  - m_1, m_2 = two masses
  - r = distance between the masses
- Einstein's general theory of relativity
  - $G_{\mu\nu} \equiv R_{\mu\nu} - \frac{1}{2} R g_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}$
  - $G_{\mu\nu}$ = Einstein tensor
  - $R_{\mu\nu}$ = Ricci tensor
  - R = curvature scalar
  - $g_{\mu\nu}$ = metric tensor
  - π = mathematical constant pi; $\pi = 3.14159265\cdots$
  - G = gravitational constant; $G = 6.67430 \frac{m^3}{kg \cdot s^2}$
  - c = speed of light; $c = 29979245 \frac{m}{s}$
  - $T_{\mu\nu}$ = energy-momentum tensor
  - (all tensors written in terms of Christoffel symbols)
- TOE (Theory Of Everything)
  - (development in progress)

3.1 Limitations

Each model has its own limitations.

In the first model ("falling down") you have to define what "down" means.

If you use the simple model "Speed of falling things" you should use, that this formula is only valid on the surface of the planet earth, at sea level at a geodetic latitude of 45°, in a perfect vacuum.

Newton's law of universal gravitation can be used for things that move with speeds close to the speed of light.

Einstein's general theory of relativity fails for very small objects (smaller than subatomic particles) where quantum effects become dominant.

The Theory of Everything is planned as a theory of gravity (and all other physical effects) that is valid always and everywhere, but scientists are still far away from their goal.

4 Properties of a good model

A good model should be:

- purposeful
- complete with respect to this purpose
- formal
- computable
- describable