

THE ROLE OF MODELING IN SOFTWARE ENGINEERING EDUCATION

Peter B. Henderson¹

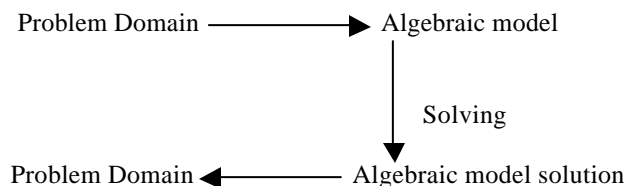
Abstract - In traditional engineering disciplines students are introduced to modeling as a tool for studying/checking the feasibility, functionality, trade-offs, features, design and construction pitfalls, cost analysis, etc. of a system. Physical and/or mathematical models are built to better understand the artifacts to be designed and constructed. In the design, development and maintenance of software systems, constructing a model is not a standard preliminary activity. This is primarily because it is not something software engineers are used to doing and there have not been sufficiently powerful software system modeling tools or techniques available. It is usually impossible to build a physical model of a software system, so any model must be mathematical in nature. The mathematical fundamentals required for constructing models of software systems will be introduced, and it will be shown how they can be used in an undergraduate software engineering or computer science curriculum to introduce students to the importance of modeling.

Index Terms – Software Engineering Education, Modeling, Discrete Mathematics, Logic.

INTRODUCTION

The noun model means “a miniature representation of something; or a pattern of something to be made”. In traditional engineering there are numerous examples and types of models. There are both physical models and mathematical models. Indeed, mathematics is a key tool for modeling in most disciplines. Evidence for this in undergraduate education comes from the recent MAA CUPM Curriculum Foundations Project [6] in which only one topic, ‘modeling’ was found to be universally significant for a wide range of disciplines - engineering, economics, computer science, physics, chemistry, biology, business, manufacturing, statistics, mathematics, etc.

The illustration below shows how students learned to use algebra as a basic mathematical modeling tool.



Note that for under, or incorrectly, specified problems an algebraic solution may not exist. Also, errors can occur when building the model, leading either to incorrect or non-existent solutions. Accordingly, it is extremely important that students learn to validate the models they build. This is a significant weakness in software engineering since model building is not usually practiced.

In traditional engineering, continuous mathematics is the primary mathematical modeling and problem-solving tool. Engineers build a model from which they use mathematical manipulations to find solutions. Some models cannot be solved using direct mathematical manipulation techniques, in which case approximate solutions are often found using a computer. An example might be the design of a wing for the next generation supersonic passenger plane. One way to check the mathematical model and resulting solution is to build a physical model and test it in a wind tunnel. Unfortunately, software engineers don't usually have this luxury.

Since software is abstract in nature, tangible models of software can't be constructed. Accordingly, software engineers are constrained to mathematical models, and until recently, automated tools for modeling complex software systems did not exist [1,3].

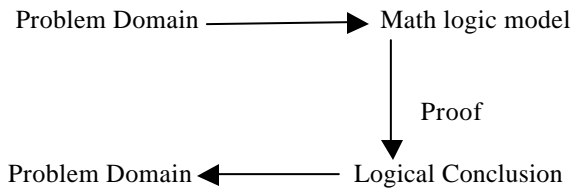
It is impossible to design and implement a correct software system without a model. So, in software engineering, what form does the model usually take? Correct, a mental model. This model is then expressed as a set of requirements and specifications. From these a system is designed using a variety of graphically oriented design modeling languages such as data flow diagrams, structure charts, statecharts, and UML. Software engineering students should also learn the rudiments of specification languages such as Z, VDM, etc. A significant component of undergraduate software engineering education is devoted to implementing software systems rather than learning and applying fundamental mathematics.

The final system is usually the first complete executable “model” of the desired artifact. Only now can it be checked against the desired mental model, the potentially inaccurate requirements/specifications, etc. – often by extensive testing. This deviates significantly from the traditional engineering process where the model is constructed and validated during the early stages. Could this be a reason why software systems are so susceptible to errors and problems? As with traditional engineering, we will see that mathematics is an important tool for modeling software systems.

¹ Peter B. Henderson, Butler University, Department of Computer Science and Software Engineering, Indianapolis, IN 46208 phenders@butler.edu

BASIC MATHEMATICAL MODELING IN SOFTWARE ENGINEERING

In software engineering discrete mathematics and mathematical logic are the primary mathematical modeling tools. Below is a very simple example from mathematical logic – a logic word problem. It uses the following strategy.



Consider the following propositional logic word problem:

Either Lucretia is forceful or she is creative. If Lucretia is forceful, then she will be a good executive. It is not possible that Lucretia is both efficient and creative. If she is not efficient, then either she is forceful or she will be a good executive. Can you conclude that Lucretia will be a good executive?

First, define appropriate propositional logic variables.

- f ≡ Lucretia is forceful
- g ≡ Lucretia will be a good executive
- e ≡ Lucretia is efficient
- c ≡ Lucretia is creative

Construct a mathematical logical model from the problem statement. The following logical statements are true:

- $f \vee c$ { inclusive interpretation of ‘or’ }
- $f \rightarrow g$
- $\sim (e \wedge c)$
- $\sim e \rightarrow (f \vee g)$

The desired conclusion: g is true

Now, ‘solve’ the model to determine if g is true. The argument can be a direct proof, or a proof by contradiction, using a sequence of valid argument forms such as modus tollens, proof by cases, rule of contradiction, etc. [2].

Finally, translate the result back into the problem domain.

g is true → Lucretia will be a good executive

This is a very simple example demonstrating declarative modeling (i.e., there is no explicit state information). Propositional logic, predicate logic, pure Prolog and functional languages are typical examples of declarative tools available for building models. There are many other basic mathematical modeling tools available to the software engineer. Most well known are grammars and regular

expressions for defining language syntax, and finite state machines, statecharts, and petri-nets for specifying state based functionality. Usually these are used to model only parts of the desired system. Another important form of modeling that will not be addressed in this paper is user interface modeling.

BEYOND BASIC MODELING TOOLS

One important property of a model is that it can be tested or checked. For software systems, this means that the model should be executable in some way. For logic, theorem provers and Prolog provide “executable” models. Compiler based front end tools can be used to create “executable” models from grammars and regular expressions. Also, numerous tools are available for creating executable finite state machines, state-charts and petri-nets.

The ideas underlying model building and checking are to be able to construct a model that can then be checked against a set of specifications for the desired system. This will expose potential flaws in the model, the specifications, or both. This process of building a model and specification leads to a better understanding of the functionality of the desired system, the lack of which is one of the primary problems with software systems development today. An added benefit is that the task of identifying higher-level abstractions when constructing a model leads to a better understanding of the potential design and implementation of the system. This is similar to learning from actually designing and implementing the system. However, due to the potential cost of revising a design/implementation, what was learned is often not incorporated into the final system.

One of the major problems is that the functionality of software systems we wish to build is very complex yielding models that incorporate a potentially astronomically number of states. Fortunately, research over the past 15 to 20 years has helped to mitigate this problem [1,3].

Numerous model building and checking systems based on this research, such as SPIN, SMV, Nu-SMV, and Asml, etc., are now available. Also, undergraduate courses will begin to evolve shortly [5]. One of the keys to the success of such courses will be the mathematical preparation of the students. Weakness in mathematics, especially in the United States, is evident from studies [9] and the content of most formal methods courses in which a significant component of the course is dedicated to introducing the foundational logic and mathematics. For this reason it is important for computer science and software engineering students to be introduced to the mathematical foundations early and to have these foundations reinforced in subsequent courses.

FOUNDATIONAL MATHEMATICS

The CCCS [7] and CCSE [8] curriculum specify 43 and 56 hours of foundational discrete mathematics respectively. In addition, in the SE curriculum recommendation there are approximately an additional 30 hours of math applications

(specifications, modeling, pre & post conditions, invariants, etc.). The fundamental discrete mathematics topics specified in SEEK are:

- Functions, Relations and Sets
- Basic Logic (propositional and predicate)
- Proof Techniques (direct, contradiction, inductive)
- Basic Counting
- Graphs and Trees
- Discrete Probability
- Finite State Machines, regular expressions
- Grammars
- Numerical precision, accuracy and errors
- Number Theory
- Algebraic Structures

It may not be immediately clear what the relevance of these mathematical concepts is to educating our students other than they are important for the software engineers' tool kit. Modeling and formal specifications are concrete applications of these concepts that can be used to motivate student learning. Indeed, many of these mathematics concepts are required foundations for modeling tools such as SMV and specification languages such as Z.

A simple example of modeling using propositional logic was presented earlier. One does not have to look far for ways in which the other concepts listed can be used to model real problems and fundamental computer science ideas. Functions are an important modeling tool for numerous problems, are central to functional based system design and both functional and imperative programming languages. Among other things, sets and relations model real world knowledge such as collections of objects (e.g., all cars in a dealers' inventory) and relationships between collections of objects (e.g., a car and its price, features, etc.). The reader can identify ways in which most of the other concepts can be used as modeling tools.

INCORPORATING MODELING INTO THE UG SE CURRICULUM

In software engineering almost everything is a model – models are pervasive. It is important for students, and faculty, to understand and appreciate this. Accordingly, the concepts of a model and modeling should be introduced early and continually reinforced. One simple technique is to use these terms frequently in all courses, especially in the introductory ones such as foundational discrete mathematics, computer science and software engineering courses. Below is an overview of some ways in which the concepts of modeling can be introduced into an undergraduate software engineering curriculum.

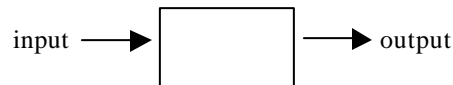
- *Discrete mathematics* – most concepts in discrete mathematics (logic, sets, functions, relations, graphs and trees, counting, etc.) are used for building, reasoning

about and analyzing abstract models. This should be emphasized through examples and exercises.

- *Computer science* – in all courses, stress that most concepts in CS are abstract and therefore only exist as models. This includes informal problem statements, data structures, flow-graphs, etc. Even algorithms and programs simply represent models of some desired computation (see the section below, Models of Computation).
- *Software Engineering* – are usually upper-division courses and students taking them are more mature. With a proper introduction to modeling and mathematical foundations in the first two years, more advance concepts such as formal specifications, mathematical model building, modeling tools, and model checking can be addressed

MODELS OF COMPUTATION

Models of computation are the heart of software systems. In its simplest form, a computation is a task performed on some input data to produce a desired output.



There are several models of computation computer science and software engineering students are introduced to. These include, sequential, conditional, iterative, recursive, deterministic, non-deterministic, parallel, concurrent, object oriented, and logic and functional. Concepts for developing, reasoning about, verifying and/or analyzing models of computation include: pre and post conditions, loop invariants, finite state machines, equational logic, and data invariants. The importance of computational models is emphasized in the CCSE Software Engineering Education Knowledge, the SEEK, list of fundamental core concepts [8].

All models of computation can be reasoned about using mathematics. As a simple example, consider the assignment rule for sequential computations [2]. This rule says that if we know the post-condition of a sequence of assignment statements, then we can determine the (weakest) precondition by using algebraic substitutions. For example,

$$\begin{aligned}
 & \{ r = 2^i \} \\
 & i \leftarrow i + 1 \\
 & \{ r = 2^{i-1} \} \\
 & r \leftarrow 2 * r \\
 & \{ r = 2^i \}
 \end{aligned}$$

Here $r = 2^{i-1}$ is obtained by substituting $2 * r$ for r in the post-condition $r = 2^i$ yielding $2 * r = 2^i$ which is simplified to $r = 2^{i-1}$.

Note that in this example the precondition and post-condition are the same. This is an important property when these statements are used in the body of an iteration for computing 2^n for $n \geq 0$. In other words, $r = 2^i$ is a loop, or iteration invariant of this iteration. This can be proven using the principle of mathematical induction [2].

Similar sections on data structures, objects, processes, data flow, concurrency, etc. could have been included to further reinforce the concept of modeling computations.

EXAMPLE OF MODELING

True story - recently I put a dollar bill in a vending machine to purchase an item. Every item was less than one dollar and when selected resulted in the same message - "use exact change only." Also, every attempt to return the dollar, as change or a bill, failed. Was this a specification/design flaw or a clever scheme for cheating customers?

A vending machine is a familiar student programming project. For such a project, simplifying assumptions, such as sufficient change is available are usually made. Without such simplifying assumptions, the problem actually becomes rather complex. Think about why!

So, why did I lose a dollar? Because someone designed the logic of the machine without a precise specification and a corresponding 'real' model to check it against. Also, I believe the designer probably felt this was a simple problem without many variations or special cases to deal with - until implementation that is. By then is it usually too late - the forest for the trees. That is, the implementation details get in the way of seeing the big picture. Even comprehensive testing failed to catch this anomaly.

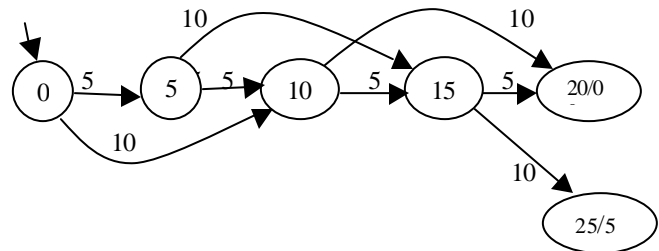
So let's construct a model first and check it against the specifications to verify the desired behavior of the system *before design and implementation*. The model building process will not only help to achieve the correct behavior, but will also provide guidance for the system design and implementation. This is an added benefit that engineers expect when they construct a physical or mathematical model.

Perhaps one of the first abstraction tools that comes to mind for modeling a vending machine is a finite state machine (FSM) whose starting state represents no coins deposited yet. As coins are inserted the FSM moves to new states representing the total deposited. To get started developing a model, let's make some simplifying assumptions:

- The machine accepts only 5 and 10 cent coins
- There is one item costing 20 cents
- The item is available
- The change maker has sufficient coins

- Once the item cost (20 cents) is reached, additional coins are not added to the total (i.e., they are bypassed to the coin return bin).

Each state in the following figure shows the total deposited and final accepting states indicate the total amount deposited and the amount of change (deposited/change). Note that once a final state is reached, no more coins can be deposited and added to the total.



As these simplifying assumptions are relaxed, one by one to more accurately model a real vending machine, the model becomes more complex. The ideas of stepwise refinement and iterative enhancement apply to model and specification development also. For the student, this complements and reinforces the application of these ideas they learned in the context of design and implementation of a software system. A similar argument holds for many other fundamental computer system design and implementation concepts such as abstraction, object oriented, refactoring, etc.

A basic formal specification of the behavior of purchasing an item from a vending machine might include the following (assuming only coins, no bills):

- The coins in the change box are valid coins
- The selected item is in the vending machine
- The total amount deposited is = cost of the item
- The change to be dispensed = $\text{total amount deposited} - \text{cost of the item}$
- The required change is can be made from the coins in the change box and those deposited. If not, return the coins deposited and inform the customer to insert exact change.
- The current total in the change box = $\text{previous total} + \text{cost of the item}$
- The number of the selected item in the machine is reduced by one (minimum possible = 0)
- The cost of the item does not change
- The vending machine is still operational for the next purchase

These can be expressed precisely in a formal specification language such as Z, VDM-SL, Larch, Resolve, etc. As noted earlier, mathematical concepts are the foundation of such languages. For example, the total amount deposited and the amount in the change box can be modeled using a

bag (a set in which the same element, e.g., nickel, dime, quarter, etc. may appear more than once).

A model checker can be used to check a model against its specification. That is, it exhaustively checks the behavior specification against the model constructed and reports any inconsistencies. For example, does the vending machine model behave as specified?

The process of validating the specifications against the model helps to identify potential misconceptions early in the lifecycle. When an incompatibility is found, either the model or the specifications, or both are incorrect. One plays off against the other leading to a better understanding of the desired behavior of the system. However, having a validated model is not sufficient due to errors of omission – behavior overlooked in both the model and specification. Often this model/specification construction and validation activity helps to identify errors of omission. After design and implemental such corrections are usually considerably more costly.

Several model construction and checking tools are available to the software engineer. These include, amongst others, SPIN, SMV, Nu-SMV, and Asml. Most are based upon finite state machines, temporal logic, and support the construction of high-level abstract component models (e.g., a change maker, item selection key pad, etc.).

THE BUG THAT KILLED

You may recall one of the most notable software errors was the Therac-25 linear accelerator machine used for cancer radiation treatment [4]. Software errors caused the machine to incorrectly compute the dosage of patient radiation resulting in at least four deaths. Would building and validating a software model first have avoided this tragedy? Perhaps, but in this case a physical model of an earlier correct version existed (it used mechanical safeguards). Unfortunately, the automated software that replaced these safeguards was not validated against this model. Lesson, just building a model is not sufficient.

The next time you rely on anything whose behavior is based upon software, be it an elevator, transportation system, voting machine, X-ray machine, electronic bank transfer, microwave, pacemaker, etc. you might ask yourself was the system behavior modeled and validated, or simply implemented and exhaustively tested.

CONCLUSIONS

Why are the concepts of a model, modeling and model checking important in an undergraduate software engineering curriculum? Software systems are abstract and only exist as models. Model building and verification will become more important in the future for software engineers as the tools and techniques mature. Modeling reinforces the important role of mathematics in the discipline. Modeling helps students to focus on the basic functionality and to learn

to reason more precisely about systems, rather than focusing on the design and implementation details.

Many computer science and software engineering graduates can't reason formally about simple algorithms (eg., give a rigorous argument that a binary search algorithm is correct). Perhaps a focus on modeling will help students to appreciate mathematics and the important role that it should play in the engineering of software systems.

ACKNOWLEDGMENT

I would like to thank Tom Hilburn and the anonymous referees for comments and suggestions that have greatly improved the quality of this paper.

REFERENCES

- [1] Clark, E.M., Grumberg, O., and Peled, D., "Model Checking," *MIT Press*, 1999
- [2] Grassman, W. K. and Tremblay, J.P., "Logic and Discrete Mathematics: A Computer Science Perspective", *Prentice-Hall*, 1996
- [3] Huth, M. and Ryan, M., "Logic in Computer Science: Modeling and Reasoning about Systems", *Cambridge University Press*, 2001
- [4] Leveson, N.G., "Software: System Safety and Computers", *Addison-Wesley*, 1995.
- [5] Liu, H. and Gluch, D.P., "A Proposal for Introducing Model Checking into an Undergraduate Software Engineering Curriculum", *16th Southeastern Small College Computing Conference*, Nov. 2002.
- [6] MAA CUPM - CRAFTY Curriculum Foundations Project, <http://www.mathsci.appstate.edu/~wmcb/CFF/>
- [7] Roberts, E., Shackelford, R, et al, "Computing Curricula 2001 : Computer Science Volume", <http://www.acm.org/sigcse/cc2001/>, Dec. 15, 2001.
- [8] Sobel, A., "CCSE Software Engineering Education Knowledge – Final Version", April 30, 2003, <http://sites.computer.org/ccse/known/FinalDraft.pdf>
- [9] Tucker, A. B., Kelemen, C.F., and Bruce, K. B., "Our Curriculum Has Become Math-Phobic!", *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, Feb. 2001, pp. 243-247.