Multi-faceted Practical Modeling Education for Software Engineering

Suresh Kothari[1] and Jeremias Sauceda[2]

[1] Electrical and Computer Engineering, Iowa State University. kothari@iastate.edu
[2] EnSoft Corp. pi@ensoftcorp.com

Abstract. It is a challenge to teach modeling to undergraduates. Primarily, the difficulty is of teaching abstract concepts because it is hard for students to digest and appreciate abstractions. This paper is about developing a curriculum in which students can experience how models enable one to: find solutions, verify solutions, and be able to experiment with possible solutions.
In this paper we present two modeling topics covered in an undergraduate course taught at Iowa State University (ISU). These topics are chosen for their practical importance. Our primary goal is to enable students to learn how to apply modeling to solve practical problems of large software and be familiar with state-of-the-art modeling tools used in industry. Model-based problem solving makes it easier for students to appreciate and learn modeling.

1 Introduction

Models help us understand how complex things work. Engineers use mathematics as a modeling tool to reason about complex systems. When Boeing designs a new jet, they model the airplane in software, where they run millions of simulations to understand how the shape of the fuselage, weight of the components, position of the cockpit, and numerous other variables affect lift and fuel efficiency.

Models enable us to abstract and focus on essentials of a problem. The abstraction becomes the basis for designing an efficient and sound solution to an otherwise difficult problem. With modeling, we can find a solution that works for not just one but a large number of seemingly different problems. The following two problems are seemingly very different but a common model works for both.

Problem 1: John is 3 years older than Jill. Together their ages add up to their mother's age. Their mother is 45 years old. How old are John and Jill?

Problem 2: A water tank holds 3 gallons of water more than another water tank. Together the two water tanks can hold 45 gallons of water. What is the capacity of each water tank?

The common model, a linear system of equations, suppresses irrelevant details. In general, a good model captures the essential core of the problem. There

are other important points about models that make them useful and powerful. Models suggest generalizations applicable to a large class of problems. In the above example, the generalization is a linear system of equations with N variables instead of just 2 variables. Models serve as the foundation for developing efficient and accurate computational methods for solving complex problems. In this paper, we discuss how to effectively bring out these modeling highlights in the context of software engineering.

The software engineering program at ISU was started in consultation with several companies. We interact with these companies and with companies that use EnSoft's products and services. Industry is interested in Model-Based Development (MBD) as a way to improve reliability of software and produce it efficiently. Industry is not interested in modeling that works only for toy problems. To be relevant to industry, modeling should be taught in the context of large real-world software. One topic, Unified Modeling Language (UML), is of great interest to industry. Since a number of textbooks already discuss UML at length, this paper focuses on other MBD topics which are often not taught in software engineering curricula but are of interest in industry.

It is a challenge to teach modeling to undergraduates. Primarily, the difficulty is of teaching abstract concepts because it is hard for students to digest and appreciate abstractions. Teaching models in a software engineering course is especially difficult. Other engineering disciplines have well established modeling techniques and tools. For example, the finite element modeling is well established and routinely taught in mechanical engineering. Software engineering itself is a relatively new and evolving discipline and the modeling of software is more so.

We present a case study based on an elective course in our undergraduate software engineering degree program at Iowa State University. After introducing a few basic concepts, modeling is primarily taught by presenting a set of problems. The problems are chosen so that the students can understand and appreciate their importance in real-world software engineering but it is difficult for students to solve the problems without applying modeling. Then modeling techniques are introduced to solve the problems. We have not yet done a systematic assessment, but we have received numerous positive comments from students, impromptu as well as comments through course evaluation at the end of the semester. Based on the feedback, we believe that this problem-based teaching makes it easier for students to appreciate and learn modeling. We present two representative examples of problem-based teaching of software modeling.

2 Multi-faceted and Practical Modeling

While the goal remains the same, to improve reliability of software and produce it efficiently, modeling is a multi-faceted activity with different types of use cases. Modeling can benefit varied software engineering tasks including:

- Gathering requirements
- Estimating time and cost for software evolution and maintenance projects
- Designing to meet real-time performance and resource constraints

- Generating software from specifications
- Analyzing software find defects, validate, and verify
- Generating test cases to meet coverage requirements
- Enabling efficient communication between clients, developers, managers, and QA personnel

In this paper we present two use cases of incorporating model-based development in a software engineering curriculum: (a)generating software from its model, and (b) model-based software analysis. We advocate these use cases because of their practical importance and two very different facets they represent of model-based software engineering.

Modeling paradigms differ depending on the use-cases. For example, The Constructive Cost Model (COCOMO) [1] is an algorithmic software cost estimation model developed by Barry W. Boehm. The model uses a basic regression formula with parameters that are derived from historical project data and current as well as future project characteristics. Our examples include: Simulink models and graphical models of program artifacts and their relationships.

One important modeling consideration is domain-specificity. The prime motivation is to make programming easy and reliable for a specific domain of applications. A good model is an abstraction that is close to the semantic core of the problem - it could be an equation, a graph or something else. A model should not be just a notation (syntax) it should enable efficient semantic processing. Semantic processing implies the ability to: find solutions, verify solutions, and be able to experiment with possible solutions. A domain is a class of problems that share a common semantic core. A generic abstraction may not be close enough to the core semantics of the domain and that necessitate the need for a domain specific language (DSL). Excel-type spreadsheet is a good example of a widely used domain specific programming environment.

We discuss Simulink models as domain-specific programming environment. Simulink is a mature modeling environment widely used in industry but rarely presented to software engineering students. Simulink has an underlying domain-specific language (DSL) and a graphical programming interface. Simulink is used specifically to develop embedded control systems software in aerospace, automotive and other companies. These companies are required to produce highly reliable software because of the safety-critical nature of their applications. The role of Simulink has shifted from a prototyping tool to a software production tool. In this process, the Simulink models have grown reaching 100K-block models. The number of lines of generated C code per Simulink block varies from one to about hundred lines. Developing and maintaining large Simulink models intrinsically requires software engineering principles and practices. The domain experts, the control systems engineers, are not typically familiar with software engineering practices to make effective use of version control systems to facilitate team work for developing large models. Companies need interdisciplinary teams of control and software engineers. A software engineer can play an effective role in such teams if he or she is familiar with the nuances of Simulink models.

Our second use-case brings out domain-specificity in a different way. We use Atlas, a generic program analysis platform from EnSoft. It offers query-based programming interface to write domain-specific analysis programs. We use this platform to teach students the idea of building domain-specific analysis models. This is possible because the analysis platform offers a query language. Generic program relationships are pre-computed by the platform and those relationships can be accessed through the query language to build domain-specific analysis scripts, for example, analysis scripts to detect malware in Android apps. Students learn and apply domain-specific knowledge of Android Manifest [23] and semantics of Android APIs [24] to write domain-specific analysis scripts.

3 Modeling Tools

Our case study involves two modeling tools: Simulink from MathWorks [2] and Atlas from EnSoft [3]. Simulink was selected because it is a widely used tool for modeling safety-critical control software. Atlas tool was selected because of its interactive analysis capability and query-based programming interface to write analysis scripts. These capabilities make it much easier to perform analysis experiments.

Simulink - Domain-specific Graphical Programming

Executable models are created graphically using Simulink. Simulink Coder™ (formerly Real-Time Workshop ®) generates and executes C and C++ code from Simulink models [5]. Simulink is typically used with other tools such as SimDiff 4 Team [6] for differencing and merging Simulink models, and Reactis [7] for testing and validating Simulink models.

There are two major classes of elements in Simulink: **blocks** and **lines**. Blocks are used to generate, modify, combine, output, and display signals. Lines are used to transfer signals from one block to another.

Blocks have zero to several input terminals and zero to several output terminals. Unused input terminals are indicated by a small open triangle. Unused output terminals are indicated by a small triangular point.

Lines transmit signals in the direction indicated by the arrow. Lines must always transmit signals from the output terminal of one block to the input terminal of another block. One exception to this is that a line can branch off of another line. This sends the original signal to each of two (or more) destination blocks. Lines can never inject a signal *into* another line; lines must be combined through the use of a block such as a summing junction. A signal can be either a scalar signal or a vector signal. The lines used to transmit scalar and vector signals are identical. The type of signal carried by a line is determined by the blocks on either end of the line.

Building a Simulink model is accomplished through a series of steps:

1. The necessary blocks are gathered from the Library Browser and placed in the model window.

2. The parameters of the blocks are then modified to correspond with the system we are modeling.
3. Finally, the blocks are connected with lines to complete the model.

Simulink models can be executed through an Interpreter. The user can go to the **Simulation** menu and click on **Start**. With more complicated systems, the user can see the progress of the simulation by observing its running time in the lower box of the model window. Videos and Examples of Simulink are available at [8].

Atlas Graphical Modeling Platform for Program Analysis

Atlas is an analysis and visualization platform to build and study graphical models of software. It is available as an Eclipse plug-in. It has a graphical query language for constructing and analyzing graphical models of software. The queries can be executed through Scala Interpreter or they can be embedded in a program analysis script written in Java or Scala.

The Atlas platform builds a database of relationships between program artifacts. The user issues queries against this database and the results are shown as graphical models. The queries are composable. The result of a query can be stored in a variable which can be passed as input to another query. The Atlas platform has built-in graph algorithms that can be used through queries to transform and traverse graph models.

In Atlas, the program analysis is decoupled from its use for modeling a particular class of applications. The advantage is that domain-specific models can be built easily. Currently, ISU and EnSoft have a joint research project funded by the DARPA APAC program to build a Security Toolbox for detecting malware in Android apps. The toolbox is being developed as a set of analysis scripts that incorporate domain knowledge of Android. Our case study was focused on operating systems so that students could use their domain knowledge to build models for XINU [9] and Linux 2.6.31 kernels. Videos and Examples of Atlas are available at [10].

4 Case Study: Examples of Teaching Modeling

The two examples are drawn from a course taught as an elective for the undergraduate Software Engineering program at ISU. The third example covered in the course, model-based testing, is not included here. We use the chapter on model-based testing from the book [14] and the tool Spec Explorer [4] from Microsoft Research.

We strive to teach modeling as a problem solving skill. We first pose a problem and let the students make an attempt to solve the problem. We tell students to think of a model to solve the problem. Then, we teach a modeling concept relevant to the problem and build a model using one of the tools we have briefly

described. To reinforce the modeling concept and the use of the tool, we give another similar but a bit more challenging problem as a modeling homework. To stay within page limits of the paper, the examples are not the complete exercises that students go through. The examples are meant to bring out how modeling can be taught through concrete problems that students can find interesting and challenging. The problems are specifically chosen so that students can appreciate the value of modeling to solve non-trivial problems.

Example I: Programming with Simulink Models

To demonstrate how Simulink can be used to investigate a real-world system, we ask students to create a simplified, first-order model of the motion of a car. If we assume the car to be travelling on a flat road, then the horizontal forces on the car can be represented by:

- v is the horizontal velocity of the car (units of m/s).
- F is the force created by the car's engine to propel it forward (units of N).
- b is the damping coefficient for the car, dependent on wind resistance, wheel friction, etc. (units of $N * s/m$) We have assumed the damping force to be proportional to the car's velocity.
- M is the mass of the car (units of kg).

Writing Newton's Second Law for the horizontal direction thus gives:

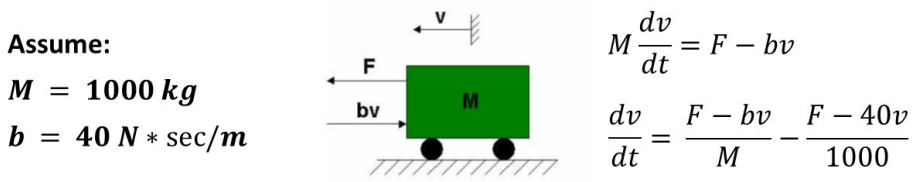

Fig. 1. Free Body Problem

To be able to successfully simulate the system, we specify an applied input, F. Let us assume the car is initially at rest, and that the engine applies a step input of $F = 400N$ at $t = 0$. This is approximately equivalent to the car's driver quickly pushing down and holding the gas pedal in a steady position starting from a stoplight.

Note that the Simulation A does not appear to show the velocity approaching a steady-state value, as we would expect for the first-order response to a step input. This result is due to the settling time of the system being greater than the 10 seconds the simulation was run. To observe the system reaching steady-state, we click **Simulation, Parameters** in the model window, and change the Stop Time to 150 seconds. Now, re-run the simulation and note the difference in the velocity graph Output B in Figure 2.

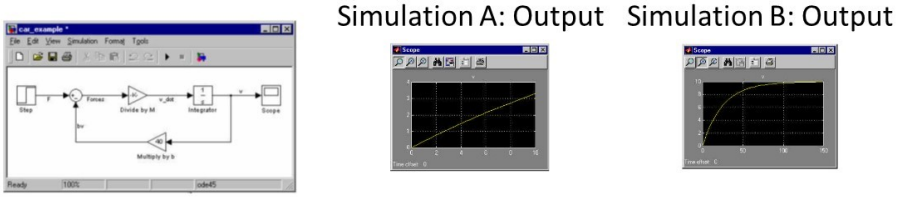

Fig. 2. Simulink Model for the Free Body Problem and Simulation Results

After engaging students in working with a couple of different Simulink models, we get into a broader discussion of following points:

- Why domain specificity is important?
- Graphical programming
- Quick prototyping and simulation
- Unifying programming and design
- Simulink models vs. C programming

Example II: Reasoning about Operating System Code by Constructing Models

We present the device driver code (`dswrite()`) shown in Figure 3 from the Xinu operating system. It includes a `getbuf()` call to allocate memory but no `freebuf()` call to deallocate memory. We ask the students to check if it is a memory leak. Students quickly go to `dskenq()` because the pointer to the allocated memory is passed to it as a parameter. Soon they find that they have to examine many functions. After examining a few functions they hit a barrier because the pointer to the memory is assigned to a global variable. Now they do not know which functions to examine because the global variable can be accessed by any function. It is time to introduce modeling to solve this problem.

```
dswrite(devptr, buff, block)
 struct devsw *devptr;
 char *buff;
 DBADDR block;
{
  struct dreq *drptr;
  char ps;
  disable(ps);
  drptr = (struct dreq *) getbuf(dskrbp);
  drptr->drbuff = buff;
  drptr->drdba = block;
  drptr->drpid = currpid;
  drptr->drop = DWRITE;
  dskenq(drptr, devptr->dvioblk);
  restore(ps);
  return(OK);
}
```

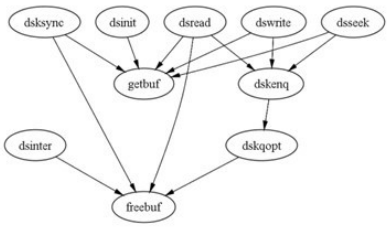

Fig. 3. Reasoning about Memory Leak Using a Graphical Model

We build a graphical model for solving the memory leak problem. The goal is to create a model to find all relevant functions and their call relationships.

Note that constructing a call graph of `dswrite` is not enough. A function that is not called directly or through a call chain can access the global variable that contains the address of the allocated memory and deallocate the memory. It can also happen that after getting the memory address the pointer may be passed through a call chain eventually to a function that calls `freebuf` to deallocate memory. So, our first model is the *reverse call graph* (**rcg**) of `getbuf` and `freebuf`. This model gives a conservatively relevant subset of functions and their call relationships.

We can build a better model by incorporating another constraint. We note that the memory was allocated for a structure of type `dreq`. The ***rcg*** can have many functions that do not *read* or *write* to variables of type `dreq`, and hence these functions are not relevant to our memory leak problem. They should not be included in the model. Our refined model can be an induced subgraph of the ***rcg*** that includes only the functions that *read* or *write* to variables of type `dreq`. It is easy to build this refined model using a sequence of Atlas queries as follows:

1. `#m1 = rcg(getbuf, freebuf) - rcg is stored in a variable #m1`
2. `#temp = ref(dreq) - temp captures the functions that read or write dreq`
3. `#m2 = #m1 intersection #temp - #m2 is the refined model`
4. `show(#m2) - the model is shown graphically`

By executing the above sequence of four queries, Atlas produces a graphical model shown in Figure 3.

A quick review of the model brings out a couple of important points. First, there is a call chain from `dswrite` to `freebuf` which means that at least on some execution paths the memory is deallocated. This model cannot tell us if the memory is deallocated on *all* execution paths, we need a different model to check that. The `dsinter` calls `freebuf` but not `getbuf`. That `freebuf` call can potentially account for missing deallocations on some of the paths. In fact, that happens to be the case in this example. We then get into other models that can facilitate such validations. We have done a project in class where students have used models to validate the Linux kernel 2.6.31 for a safe synchronization property which involves analysis quite similar to checking memory leaks. The models used for the Linux kernel validation were originally developed by two graduate students as a part of PhD research. At the end of the project the students write a report classifying all instances memory allocations according to the results of their analysis and for a few selected instances they provide a detailed description of: the analysis they performed, the relevance of the tool in performing the analysis, and the learning outcome.

We have developed a new teaching module to show how models suggest generalizations applicable to a large class of problems. This module defines a notion of 2-event models and shows its broad applicability to a large class of problems. We show that the model is applicable for analyzing memory leaks, mismatches of mutex locks and unlocks, and interestingly to detecting malware in Android apps. The applicability to detect Android malware is a research project funded by DARPA APAC program [13]. This teaching module poses malware detection problems. The module teaches strategies to characterize and

analyze malware through graph models. The goal is to extract graphical models of program relationships and show that it corresponds to a model of malware.

The Figure 4 shows a model of malware and its correspondence to a model extracted from the software. In this model, images captured by camera are first passed as exception objects, then as asynchronous messages using Android APIs, and finally leaked to a suspicious URL. The software model is extracted using a sequence of Atlas queries.

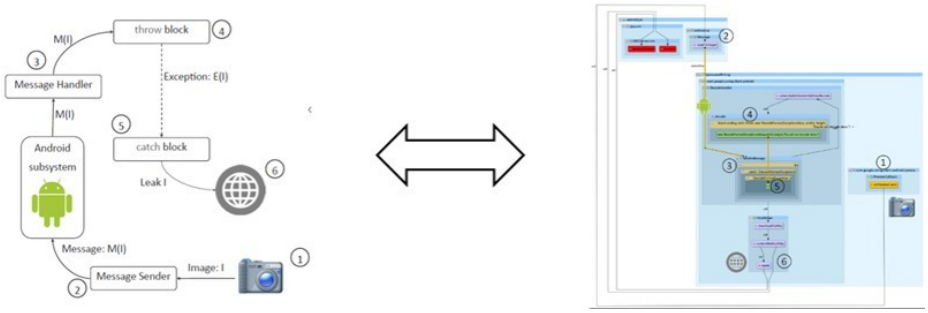

Fig. 4. A model of malware and its correspondence to a graphical model of software

5 Related Work

Three thematic groups of paper submissions were discussed in the Educators Symposium of the MODELS conference in 2008 [16]. There are also a number of other papers written about teaching model-based software development [17–20]. This paper presents a different theme that emphasizes multi-faceted practical modeling.

Pareto in [19] discussed a course in which educators give students two existing domain specific languages using Microsoft DSL toolkit. While Pareto focused on graphical domain-specific languages, and the translations from graphical languages to embedded target platforms. We focus on Simulink which can be thought of as a graphical domain-specific language.

Kramer in [21] has called abstraction the "*key skill*" in computing. Roberts in [22] refers to two aspects: removing unnecessary details and the process of generalizing concepts and finding patterns. Our case study present concrete examples of problem solving that include these aspects.

6 Conclusions

This paper advocates that modeling be taught as problem solving exercises so that students can get over the hump of abstraction and find modeling interesting and useful. We present a case study that includes two very different facets of modeling. We believe rightly taught modeling can transcend software engineering and help students learn how to think clearly. While we have not done a formal assessment of effectiveness of our approach to teach model-based software engineering, we have received many student comments through course and instructor

evaluation forms indicating that they found the topics thought provoking and useful.

Acknowledgment

This work was partly supported by MathWorks curriculum development grant and a DARPA research grant under agreement number FA8750-12-2-0126. We thank the reviewers for careful reviews and many good suggestions.

References

1. COCOMO model. http://en.wikipedia.org/wiki/COCOMO
2. Simulink. http://www.mathworks.com/products/simulink/
3. Atlas. http://www.ensoftcorp.com/atlas/
4. Spec Explorer. http://research.microsoft.com/projects/specexplorer/
5. Simulink Coder. http://www.mathworks.com/products/simulink-coder/
6. SimDiff 4 Team. http://www.ensoftcorp.com/simdiff/
7. Reactis http://www.reactive-systems.com/papers/bcsf.pdf
8. Simulink examples and videos. http://www.mathworks.com/products/simulink/examples.html
9. XINU. http://en.wikipedia.org/wiki/XNU
10. Atlas examples and videos. http://www.ensoftcorp.com/atlas/
11. spec#. http://research.microsoft.com/en-us/projects/specsharp/
12. Asml. http://research.microsoft.com/en-us/projects/asml/
13. DARPA APAC Program. http://www.darpa.mil/Our_Work/I2O/Programs/Automated_Program_Analysis_for_Cybersecurity_(APAC).aspx
14. Page, A., Johnston, K., Rollison, B.: How We Test Software at Microsoft®. O'Reilly (2009)
15. Google Talk: Automated Model-Based Testing of Web Applications. http://www.youtube.com/watch?v=6LdsIVvxISU
16. Smialek, M.: Promoting software modeling through active education. In: In Proceedings of the Educators Symposium at the 11th International Conference, MODELS 2008. (2008) 7
17. Gokhale, A.S., Gray, J.: Advancing model driven development education via collaborative research. In: Educators' Symposium, Citeseer (2005) 41
18. Cowling, A.J.: Modelling: a neglected feature in the software engineering curriculum. In: Software Engineering Education and Training, 2003.(CSEE&T 2003). Proceedings. 16th Conference on, IEEE (2003) 206–215
19. Pareto, L.: Teaching domain specific modeling. In: Symposium at MODELS 2007.
20. Experiences of teaching model-driven engineering in a software design course
21. Kramer, J.: Is abstraction the key to computing? Communications of the ACM **50**(4) (2007) 36–42
22. Roberts, P.: Abstract thinking: a predictor of modelling ability? (2009)
23. Android Manifest:. http://developer.android.com/guide/topics/manifest/manifest-intro.html
24. Android API:. http://developer.android.com/guide/topics/manifest/uses-sdk-element.html