Automating Exercise Generation: A Step towards Meeting the MOOC Challenge for Embedded Systems

Dorsa Sadigh
UC Berkeley
dsadigh@berkeley.edu

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

Mona Gupta
UC Berkeley
monagupta@berkeley.edu

ABSTRACT

The advent of massively open online courses (MOOCs) poses several technical challenges for educators. One of these challenges is the need to automate, as much as possible, the generation of problems, creation of solutions, and grading, in order to deal with the huge number of students. We collectively refer to this challenge as *automated exercise generation*. In this paper, we present a step towards tackling this challenge for an embedded systems course. We present a template-based approach to classifying problems in a recent textbook by Lee and Seshia, and outline approaches to problem and solution generation based on mutation and satisfiability solving. Several directions for future work are also outlined.

Keywords

Embedded systems education, massively open online courses, problem generation, solution generation, automated grading and feedback, exercise generation

1. INTRODUCTION

University education is in the midst of a sea change, with the rise of *massively open online courses* (MOOCs) [6]. Starting with the widely-publicized online courses at Stanford University, several universities are engaged in offering online versions of regular courses, through companies such as Coursera [1] and Udacity [3] and initiatives such as edX [2]. In addition to creating an easy access to a standard curriculum, MOOCs allow students to learn at their own speed and learn from each other using online social networking tools. Enrollments of several tens of thousands of students have been reported for some courses. This massive scale poses significant technical challenges. One of these challenges centers on *homework exercises and exam problems*: creating a large and diverse set of problems of varying difficulty, automatically grading them, preventing cheating, and providing customized feedback and new practice problems to students.

This paper describes our first steps towards addressing this challenge for an embedded systems course. The context for our work is the undergraduate course on Embedded Systems at UC Berkeley [12] and its supporting textbook [14]. We summarize here some salient points about the course and textbook made in a paper at the 2010 edition of this workshop [13]. The course and accompanying book take a *cyber-physical systems* approach to embedded systems education. They focus on how to model and design the joint dynamics of software, networks, and physical processes, specify properties thereof, and verify these properties. In contrast, many other courses and books focus instead on specific mechanisms such as interrupt controllers, memory architectures, assembly-level programming device driver design, network interfaces, and scheduling strategies. Accordingly, the textbook [14] is organized into three major parts: *Modeling*, *Design*, and *Analysis*. Modeling is the process of gaining a deeper understanding of a system through imitation. Models specify **what** a system does. Design is the structured creation of artifacts. It specifies **how** a system does what it does. Analysis is the process of gaining a deeper understanding of a system through dissection. It specifies **why** a system does what it does (or fails to do what a model says it should do). The textbook includes several exercises to help students learn the basics of modeling, design, and analysis of embedded systems. Students taking the course at UC Berkeley not only solve these homework exercises from the textbook, but also concurrently perform "hands-on" laboratory assignments and projects — this interplay between the textbook material and the lab work is described in an accompanying paper at this workshop [15].

In this paper, we address the three problems of automatically *generating new problems*, automatic *solution generation*, and *auto-grading*. Automatic problem generation involves the following task: given a sample problem, generate a new problem that is of "similar difficulty" to the sample. Thus, problem generation is fundamentally about synthesis. Solution generation is the task of automatically solving a mathematical formalization of a given problem. If one formalizes the problem in a suitable logical theory, a solution generator becomes a decision procedure for that logic. Finally, auto-grading involves checking whether a candidate solution is indeed a valid solution to the given problem. We collectively refer to these three problems as *exercise generation*.

Consider first the task of automatic problem generation. It is unrealistic and also undesirable to completely remove the instructor from the problem generation process, since this is a creative process that requires the instructor's input to emphasize the right concepts. However, some aspects of

problem generation can be tedious for an instructor, and moreover, generating customized problems for students in a MOOC is impossible without some degree of automation. Additionally, creating many different versions of a problem can be effective at reducing cheating by blind copying of solutions.

Examining problems from all three parts of the Lee and Seshia textbook, we take a *template-based approach* to automatic problem generation. Specifically, several existing exercises in the book are shown to conform to a template. The template identifies common elements of these problems while representing the differentiating elements as parameters or "holes". In order to create a new problem, the template essentially must be instantiated with new parameter values. However, it is often useful to create new problems that are "similar" in difficulty to existing hand-crafted problems. To facilitate this, we generate new problems using a bounded number of *mutations* to an existing problem, under suitable constraints and pruning to ensure well-defined results. An instructor can then select results that look reasonable to him or her.

Automatic solution generation and auto-grading are, at least in theory, somewhat easier problems. For several of the problems in the textbook that we examined, the solution generation and auto-grading problems reduce to particular classes of decision problems often encountered in formal verification, and existing techniques based on model checking [7], Boolean satisfiability (SAT) solving [17], and satisfiability modulo theories (SMT) solving [4] can be employed.

Automatic exercise generation has been studied previously for pedagogy in mathematics. Jurkovic [11] presented an approach based on instantiating parameters of algebra problems with random constants. Singh et.al [18] propose a solution for generating algebra proof problems in a three step fashion of query generation, query execution and query tuning, where the first two steps are done automatically and query tuning is done by a human expert. However, there is no previous published work on automatic exercise generation for embedded systems.

In the rest of this paper, we describe our approach for two types of problems: (i) those relating to the design and verification of extended finite-state machine (FSM) or modal models (Sec. 2), and (ii) those relating to real-time scheduling (Sec. 3). We conclude with an outline of future directions in Sec. 4.

We make one final note before diving into the details. While the setting of MOOCs provides a strong motivation for automating exercise generation, any automated techniques will also prove useful in the traditional course setting. In fact, it is in the latter setting that the ideas described herein will be tested in the Fall 2012 offering of the Berkeley undergraduate embedded systems class.

2. MODEL-BASED PROBLEMS

Model-based problems, also referred to herein as *state-machine problems*, study the interaction between three entities:

- **Models**: These are extended finite-state machines, presented either as a single automaton or as a composition of components modeling the system and its environment;
- **Properties**: These are the specifications that the model must satisfy, given in mathematical logic, or as automata, or even in (structured) natural language (English text), and
- **Traces**: The traces are either witnesses, demonstrating desired behavior, or counterexamples, showing undesirable behavior of a model, given a property.

In the remainder of this section, we present a few examples of model-based problems in Lee and Seshia [14] and explain how many of these problems can be generalized into a single template. Given this template, it can be instantiated in several ways to obtain different kinds of model-based problems. We discuss how these variants can be automatically solved and graded. Moreover, given an existing problem from the textbook, we illustrate how one can use a mutation-based approach to generate similar problems.

2.1 Generalizing Problem Instances to Templates

We surveyed exercises of the textbook [14] and noticed that a large number of them address an interaction between models, specifications and traces. Generalizing the form of this interaction, we sought to generate a template such that any model-based problem from the book will be an instance of this template. To do problem generation, we would need to choose a fresh instance of this template that is created based on a given problem. However, drawing any instance of this template blindly is not a sufficient answer for automatic problem generation. We would also wish to define a measure of difficulty since a sample instance of a generated template can be placed in a range from trivially-easy problems to unintentionally-difficult ones. The example below from Chapter 3 of [14] is an illustration of a model-based problem.

Example 2.1. *This problem considers a* ***Finite State Machine*** *that models arrivals of pedestrians at a crosswalk. We assume that the traffic light at the crosswalk is controlled by the FSM shown in the Figure 1. In all cases, assume a* ***time triggered*** *model, where both the* ***pedestrian model*** *and the* ***traffic light model*** *react once per second. Assume further that the composition of the two models is a* ***synchronous composition****.*

For the given ***pedestrian model*** *(shown in Figure 2), find a* ***trace*** *whereby* "a pedestrian arrives but the pedestrian is never allowed to cross." *That is, at no time after the pedestrian arrives is the traffic light in state red.*

This example examines a composition of a system model and an environment model. Furthermore, it asks for a trace such that a given specification holds on that trace. Based on this example and others similar to it in the textbook, we generate the following template:

Template 2.1. *Given a model* $\langle M \rangle$ *[composed of synchronous/asynchronous compostion of a system model* $\langle S \rangle$ *and an environment model* $\langle E \rangle$ *assuming the models are time*

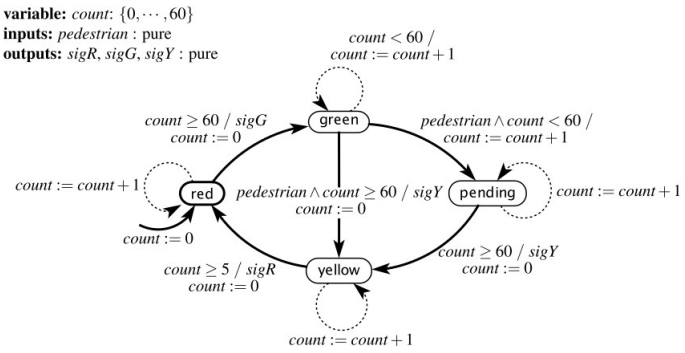

Figure 1: Traffic Light Model

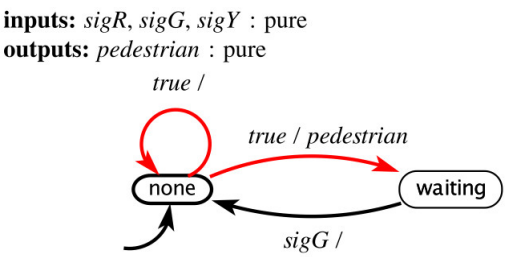

Figure 2: Two State Pedestrian Model

triggered/event triggered], and given a property $\langle\phi\rangle$ *in LTL /English. Find a trace* $\langle\psi\rangle$ *that violates* $\langle\phi\rangle$ */ satisfies* $\langle\phi\rangle$ *.*

Any instance of this template, with fresh values of $\langle M\rangle$ ($\langle E\rangle$ or $\langle S\rangle$ if the problem is directly based on a composition M of S and E), $\langle\phi\rangle$ or $\langle\psi\rangle$ generates a new problem. Also, it is possible to toggle other keywords of each of these three entities to create variations of the same problem. Some of the keywords of models mentioned in this template are whether there is a synchronous or asynchronous compostion, or if the model is time triggered or event triggered. The specification $\langle\phi\rangle$ can be given in *linear temporal logic* or in English instructions. The trace $\langle\psi\rangle$ is also either a witness that satisfies the property specified by $\langle\phi\rangle$ or it is a counterexample that proves the property $\langle\phi\rangle$ is not satisfied.

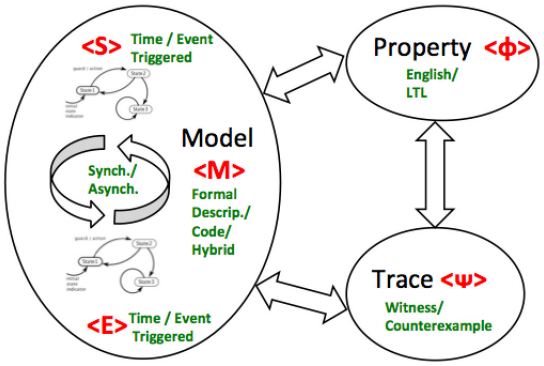

Figure 3: Models, Properties and Traces

Given	Find	Variations	Exercise #
$\langle\phi\rangle$	$\langle M\rangle$	(i)$\phi \in$ English or LTL (ii)use hybrid systems for M (iii)Modify pre-existing M	3.1, 3.2, 3.3, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.8, 9.4, 9.6, 13.2, 13.3
$\langle M\rangle$	$\langle\psi\rangle$	(i)reachable trace (ii)describe output	3.3, 3.5, 4.2
$\langle M\rangle$	$\langle\phi\rangle$	Models can be given in code or formal description	3.2, 12.3
$\langle M\rangle$ & $\langle\psi\rangle$	$\langle\psi\rangle$	Given input trace $\rightarrow$ find output trace	9.5
$\langle M\rangle$ & $\langle\phi\rangle$	$\langle\psi\rangle$	Find counterexample or witness trace	3.4, 4.3, 12.1

Table 1: Classification of Model-Based Problems in Lee and Seshia [14], First Edition, Version 1.06

After investigating the exercises from certain relevant chapters (Ch. 3,4,9,12,13) of Lee and Seshia [14], we found that more than 60% of problems fit into the model-based category, where there is a relation between models, properties and traces that fits Template 2.1. Figure 3 is an illustration of the three entities, and their characteristics. At any point, given one or two of these entities, we can ask about instances of the unknown entity. Table 1 groups exercises into different classes based on what is given and what is to be found. Each group represents an interaction between models, properties and traces. The first column shows the given entity, and the second column is the unknown entity. The third column shows some of the variations of the same class of problem. Example 2.1, fits into the last row since the model of traffic light and pedestrian are given, and the student is asked to find a trace $\langle\psi\rangle$ that satisfies some given property $\langle\phi\rangle$ described in English. In Example 2.1, model $\langle M\rangle$ is a synchronous composition of two other models $\langle S\rangle$ and $\langle E\rangle$ which are given in Figure 1 and Figure 2. Based on the construction of Template 2.1, we can assume that a template can be built for all other categories of Table 1 by taking the same approach.

2.2 Automatic Solving and Grading

Looking back at Table 1, we first attempt to find a solution technique for each category defined in that table (i.e., each row). Since model-based problems are intended to teach some of the major concepts of analysis, synthesis and verification, it is not unexpected to find that these exercises are simplified versions of well-defined research problems: *synthesis* from high-level specifications, *repair*, *simulation*, *specification mining*, and *model checking*. Several techniques and tools presented in conferences such as CAV, EMSOFT, TACAS, HSCC, DAC, PLDI, etc. can be leveraged to solve these problems.

Given	Find	Solution Technique
$\langle\phi\rangle$	$\langle M\rangle$	Constrained Synthesis or Repair
$\langle M\rangle$	$\langle\psi\rangle$	Simulation of Model
$\langle M\rangle$	$\langle\phi\rangle$	Specification Mining
$\langle M\rangle$ & $\langle\psi\rangle$	$\langle\psi\rangle$	Simulation with Guidance
$\langle M\rangle$ & $\langle\phi\rangle$	$\langle\psi\rangle$	Model Checking

Table 2: Techniques to Find Solutions for Model-Based Problems

Table 2 states a solution technique for each problem category listed in Table 1. One concern might be that techniques such as automata-theoretic synthesis or model checking, in spite of the many advances, are computationally expensive to run as auto-graders. However, it is important to note that the textbook problems are usually small and their size can be limited to within the capacity of existing tools. For instance, going back to the traffic light example, we know that the state space of traffic light and pedestrian models are fairly small; therefore, developed tools can easily handle solving the *model checking* problem described in Example 2.1. Specifically, translating the pedestrian and traffic light models as Promela Source Code for the SPIN model checker, and the specificaion as an LTL formula $\phi = \mathbf{G}(\text{pedestrian} \rightarrow \neg(\mathbf{F}\ \text{SigR}))$, SPIN easily finds a trace that satisfies ϕ. The solution is a trace that starts from (Red, None) and makes a transition to (Green, Waiting).

Automatic grading is simpler than finding a solution. For example, for a problem requiring designing a model from a specification, the solution technique involves verifying that the model satisfies the stated specification. For a problem requiring verifying if a model satisfies a stated property, if the answer is negative, the auto-grading problem simply involves simulating the stated counterexample on the model and monitoring if the property is satisfied.

2.3 Problem Generation

We now discuss how new problems similar to existing ones can be generated by modifying various elements of existing problems — the model $\langle M\rangle$, the specification $\langle\phi\rangle$, or the trace $\langle\psi\rangle$.

2.3.1 *Modifying* $\langle M\rangle$

Based on Table 1, all the categories except for the first row (constrained synthesis) require a given model $\langle M\rangle$. Therefore, one important way of generating new problems is to create new versions of models in existing problems. For instance, we can generate a new problem based on Example 2.1 by only changing the composed model $\langle M\rangle$ (or $\langle E\rangle$ or $\langle S\rangle$). In the case of Example 2.1, it is sufficient to give a variation of the pedestrian model $\langle E\rangle$. A possible variation is to change the pedestrian model from the state machine shown in Figure 2 to the state machine in Figure 4. This change increments the number of states of $\langle E\rangle$ by one, and it will further create new transtions.

Based on Example 2.1 and other similar examples, we believe that building new models generally involves making small changes to the original model. Creating these small changes can be achieved by mutating the first model using some *mutation operator*. Below, we list a sampling of *mutation operators* we have explored. Some of these operators are defined by Hierons and Merayo in [9]. In addition to those,

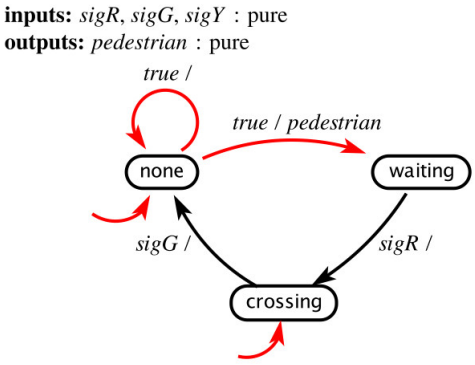

Figure 4: Three-State Pedestrian Model

we have added incrementing or decrementing the number of states, which potentially creates new transitions.

1. Changing the initial state
2. Changing the target state of a transition
3. Creating a new transition
4. Incrementing or decrementing the number of states by one

The *mutants* generated by these *mutation operators* can serve as fresh models for new problems. The difficulty of these new models are directly affected by the number of mutations made on the original model $\langle M\rangle$. To create small changes, we only consider the n^{th} *order mutant* (as defined in [9]), where the new model is created after a sequence of n mutation operators. The value of n is a measure of the change in difficulty. Although restricting the number of mutations n does not guarantee the level of difficulty of the new models, this approach eliminates a large number of trivial or unintentionally complex models. We also prune out more mutants by eliminating *non-receptive* models. (A state machine is receptive if, for each state, there is at least one defined transition on each input symbol [14].)

Using this bounded mutation approach, we have created fresh models based on the original $\langle M\rangle$. In Example 2.1, by mutating the pedestrian model with combinations of the first three mutations listed above, it is possible to create 32 new, receptive models. These models are generated using up to five mutations, from 1st order to 5th order. They are summarized in Table 3. For this particular model, five different elements of the model can be mutated — the four possible transitions and the one choice of initial state — and so the space of mutations is relatively small and can be enumerated within a second. As a model becomes more complex, the size of a model's mutation space will increase accordingly, and a goal-directed search will need to be performed. For all models, there is a saturation point at which no more receptive models may be produced using the given types of mutations. However, in order to generate interesting problems, the mutants should also model plausible scenarios. For this example, manual inspection reveals that slightly under one-third of the generated mutants are able to plausibly model a real-world scenario. These numbers are listed in the last column of Table 3. The vast majority of these mutants were produced by one or two mutations, much

lower than the saturation limit. Given a fixed set of specifications or traces, and the generated mutants, we can easily create fresh problems from different categories of Table 2.

Num. of Mutations	Num. of Models Created	Types of Mutations (number)	Num. of Realistic Models
1	4	Adding a new transition (4)	4
2	8	Combination of adding new transitions and changing initial state(4) Changing end points of two transitions (4)	4
3	8	New transition, changing initial state, changing end point (4) New transition, changing end points of two transitions (4)	2
4	8	New transition, changing initial state, changing end points of two transitions (4) New transition, changing end points of three transitions (4)	0
5	4	New transition, changing initial state, changing end points of three transitions (4)	0
6+	0	None	0

Table 3: Mutants for Pedestrian Model in Fig. 2

One of the generated mutant pedestrian models is shown in Fig. 5. The initial state has been changed and a self-loop added to the "waiting" state indicating that a pedestrian can continue to wait even when the walk signal turns green.

2.3.2 *Modifying* $\langle\psi\rangle$

In addition to changing the model $\langle M\rangle$, we would like to modify a given trace $\langle\psi\rangle$. Having a fixed model, by changing some of the traces, we can create new *simulation* problems. Example 2.2 is another exercise from Chapter 3 of [14] that describes a *simulation* problem.

Example 2.2. *Consider the state machine in Figure 6. State whether each of the following is a behavior for this machine. For readability, absent is denoted by the shorthand a and present by the shorthand p.*
(a) $x = (p, p, p, p, p, \dots), y = (0, 1, 1, 0, 0, \dots)$
(b) $x = (p, p, p, p, p, \dots), y = (0, 1, 1, 0, a, \dots)$
(c) $x = (a, p, a, p, a, \dots), y = (a, 1, a, 0, a, \dots)$

In this case, to generate new problems, we perform random simulation on the model shown in Figure 6. The output of a random simulation is a set of traces Ψ. For any $\psi \in \Psi$ we

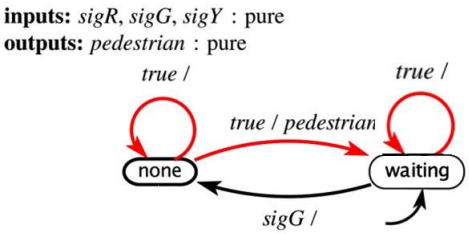

Figure 5: Mutant Two-State Pedestrian Model. The model differs from Fig. 2 in the initial state and the extra self-loop in the waiting state.

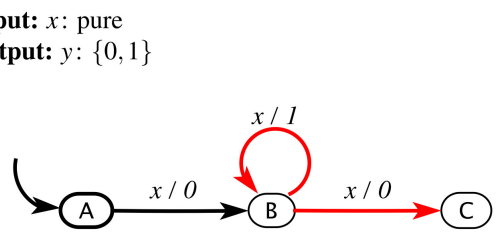

Figure 6: Example 2.2 Simulation Model

genereate ψ', where ψ' is created by a bounded number of modifications on inputs and outputs of ψ. Then, the same question about *simulation* of model $\langle M\rangle$ can be asked given ψ' instead of ψ. In Example 2.2, assume ψ is the trace given in (a) $\psi = \{x = (p, p, p, p, p, \dots), y = (0, 1, 1, 0, 0, \dots)\}$. By only modifying the fifth output of ψ from 0 to absent, we can generate $\psi' = \{x = (p, p, p, p, p, \dots), y = (0, 1, 1, 0, a, \dots)\}$. This new trace is in fact the second trace given in part (b) of Example 2.2. So we can conclude that modifying traces can easily generate new problems about the same fixed model $\langle M\rangle$.

2.3.3 *Modifying* $\langle\phi\rangle$

The last entity that we would like to modify in the model-based problems are specifications. Several model-based design problems involve giving the student a specification and asking for a model that satisfies the specification. These problems are the first type listed in Table 1, categorized as constrained synthesis or repair problems.

Modifying the specification ϕ in an existing problem can generate new problems in this category. Example 2.3 from Chapter 4 of [14] illustrates a synthesis problem where the specification ϕ is given as an English instruction in two different parts.

Example 2.3. *Automobiles today have the features listed below. Implement each feature as an extended finite-state automaton.*
(a) The dome light is turned on as soon as any door is opened. It stays on for 30 seconds after all doors are shut.
(b) Once the engine is started, a beeper is sounded and a red light warning is indicated if there are passengers that have not buckled their seat belt. The beeper stops sounding after 30 seconds, or as soon the seat belts are buckled, whichever is sooner. The warning light is on all the time the seat belt

is unbuckled.

In this example, the specification can be translated to temporal logic, specifically linear temporal logic (LTL). By modifying the temporal logic formula, one can generate variants of the problem.

Consider part (a). Let $D_i Open$ indicate that the ith door is open, and $light_{on}$ to indicate that the dome light is turned on. Assume each step corresponds to one second. Then, the property in (a) can be specified as follows:

$$\mathtt{G}\big[(D_1 Open \vee D_2 Open \vee D_3 Open \vee D_4 Open) \rightarrow$$

$$(light_{on} \wedge \mathtt{X} light_{on} \wedge \ldots \wedge \mathtt{X}^{30} light_{on})\big]$$

We can generalize this property into a template of the form:

$$\mathtt{G}\big[\phi \rightarrow (\bigwedge_{i=0}^{k} \mathtt{X}^k \psi)\big]$$

where ϕ, ψ and k are parameters to be instantiated with some user guidance.

For example, the instructor could modify the LTL formula for part (a) to model a car-alarm feature, by changing ϕ to a formula $Locked \wedge DoorOpenAttempt$, change k to 600, and ψ to a new proposition $alarm_{on}$, thus indicating the property that if the car is locked when someone attempts to open the door, the car alarm goes off and continues for 5 minutes. This problem variant is essentially the same as the existing problem (and of similar difficulty), but it is quite a different property from the application perspective.

3. REAL-TIME SCHEDULING PROBLEMS

These problems study different scheduling strategies for real-time systems. They generally ask for an unknown variable (execution time, period, deadline, etc.) given all the other information for a specific scheduling strategy with defined properties.

As in the previous section, we begin with a few example scheduling exercises from Chapter 11 of Lee and Seshia [14], and generalize these to a template. We describe how solutions can be automatically generated and graded. We also describe how new problems can be generated from the template. In both cases, the common thread is to use satisfiability solving.

3.1 From Problem Instances to Templates

We begin with an exercise from Chapter 11 of [14] that demonstrates our approach:

Example 3.1. *This problem studies **fixed priority** scheduling. Consider **two tasks** to be executed **periodically** on a **single** processor, where task1 has period $p_1 = 4$ and task2 has period $p_2 = 6$. Let the execution time of task1 be $e_1 = 1$. Find the maximum value for the execution time e_2 of task2 such that the **rate monotonic schedule** is **feasible**.*

Next, we generalize from this example to a template in the same fashion that Template 2.1 was built.

Template 3.1. *Consider $\langle n \rangle$ **periodic/sporadic** tasks, with execution times $\langle e_{1,\ldots,n} \rangle$, periods $\langle p_{1,\ldots,n} \rangle$, and (optionally) deadlines $\langle d_{1,\ldots,n} \rangle$ with **fixed/dynamic** priorities, **with /without** a precedence graph, on $\langle m \rangle$ processors. Given values for some of the parameters, and a choice of scheduling strategy $\langle S \rangle$ [RM,EDF,...] compute (max/min) values for the remaining parameters so that the schedule satisfies a desired property (is feasible, or achieves a particular processor utilization, etc.).*

(Here RM and EDF stand for "rate-monotonic" and "earliest deadline first" respectively.)

Not every instance of Template 3.1 can be taken as a meaningful real-time scheduling problem. Several constraints between the parameters and keywords of this template need to be satisfied in order to generate *well-formed* and *non-trivial* problems. These constraints depend on the scheduling strategy we intend to use.

Example 3.2 is a slight variation of Template 3.1 where *Earliest Deadline First* scheduling strategy is studied. The only difference between this example and Example 3.1, other than the desired scheduling strategy, is having dynamic priorities and introducing deadlines. It is easy to imagine how a simple constrained mutation-based procedure can generate Example 3.2 from Example 3.1. In fact, Example 3.2 is the second exercise from Chapter 11 of [14] (Version 1.06).

Example 3.2. *This problem studies **dynamic-priority** scheduling. Consider **two tasks** to be executed **periodically** on a **single** processor, where task 1 has period $p_1 = 4$ and task 2 has period $p_2 = 6$. Let the deadlines for each invocation of the tasks be the end of their period. That is, the first invocation of task 1 has deadline 4, the second invocation of task 1 has deadline 8, and so on.*
*Let the execution time of task 1 be $e_1 = 1$. Find the maximum value for the execution time e_2 of task 2 such that **EDF** is **feasible**.*

3.2 Automatic Solving and Grading

Finding a solution for real-time scheduling problems depend on the given scheduling strategy. Closed-form formulas relating the various parameters for different scheduling strategies can be obtained from books and monographs on the subject, such as the excellent book by Buttazzo [5].

Thus, to automatically solve a scheduling problem, we only need to encode the formula for a solver that can handle the operators appearing in it. Satisfiability modulo theories (SMT) solvers [4], which are SAT-based theorem provers, have made tremondous progress over the past decade, and can be used as back-end solvers.

An SMT solver finds a satisfying assignment for the unknown parameters given all the other parameters in the statement of problem. For instance, to find a solution for Example 3.1, we need to analyze rate monotonic schedules. Based on Liu and Layland's theorem [16] (shown as Theorem 1), we derive Equation 1, which describes the relation between periods and maximum execution times of a rate monotonic scheduler with two tasks.

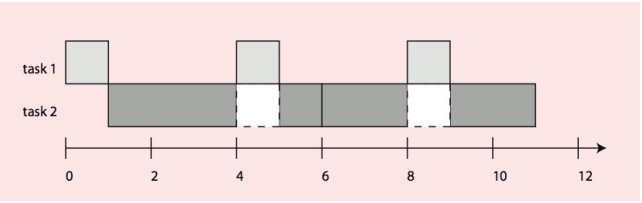

Figure 7: Rate Monotonic Schedule for Example 3.1

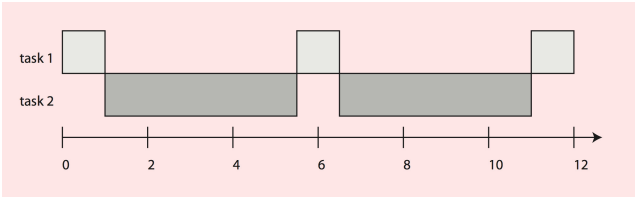

Figure 8: Earliest Deadline First for Example 3.2

Theorem 1. *Given a preemptive, fixed priority scheduler and a finite set of repeating tasks $T = \tau_1, \tau_2, \ldots, \tau_n$ with associated periods $p_1, p_2, \ldots, p_n$ and no precedence constraints, if any priority assignment yields a feasible schedule, then the rate monotonic priority assignment yields a feasible schedule.*

Formulating Equation 1 as an SMT problem, and feeding it to an SMT solver will result in a correct satisfying solution for maximum execution time of the second task e_2. This schedule is illustrated in Figure 7.

$$[p_1 < p_2] \rightarrow \lceil \frac{p_2}{p_1} \rceil * e_1 + e_2 = p_2 \tag{1}$$

$$[4 < 6] \rightarrow \lceil \frac{6}{4} \rceil * 1 + e_2 = 6 \implies e_2 = 4$$

The equation for rate monotonic schedules can be generalized for n tasks, as shown in Equation 2. Here we assume an ordering on the periods of tasks such that $p_i < p_{i+1}$ (p_i represents the period of τ_i).

$$[p_1 < \cdots < p_n] \rightarrow \lceil \frac{p_n}{p_1} \rceil * e_1 + \cdots + \lceil \frac{p_n}{p_{n-1}} \rceil * e_{n-1} + e_n = p_n \tag{2}$$

The same idea can be applied to other scheduling strategies using corresponding formulas. For instance, we will have a different formula for solving Example 3.2. Equation 3 shows this formula, which corresponds to an *Earliest Deadline First* scheduling strategy with *dynamic priorities* for two tasks. This schedule is shown in Figure 8.

$$\frac{e_1}{p_1} + \frac{e_2}{p_2} = 1 \tag{3}$$

$$\frac{1}{4} + \frac{e_2}{6} = 1 \implies e_2 = \frac{9}{2} = 4.5$$

To summarize: the solution approach for real-time scheduling problems satisfying Template 3.1 is as follows: given a parameterized formula for the desired scheduling strategy, in order to compute a solution, we plug in values for the known parameters and solve for the unknown parameters using an SMT solver.

Auto-grading for this class of problems is also very easy: we simply plug in the values for parameters into the formula and check that the identity is satisfied.

3.2.1 Implementation Details

We model the formulation for the desired scheduling strategy in SMT format, and represent the integers in formulations as bit vectors. We use finite-precision bit-vector arithmetic for two reasons: (i) to bound the search to small values a student can work with, and (ii) to handle non-linear operations such as division, for which there good solvers handling integer arithmetic do not exist. Using Beaver (an SMT Solver for Bit-Vector Arithmetic) [10], we were able to find a solution for execution time of the second task e_2 in 0.073 seconds for the *rate monotonic* schedule (Example 3.1), and in 0.346 seconds for the *Earliest Deadline First* scheduling strategy (Example 3.2).

3.3 Problem Generation

In addition to automatically solving real-time scheduling problems, we would like to automatically create new ones. Problem generation for this group of exercises is relatively more straightforward than for "design" problems that involve generating a model from a specification. Using the same template-based approach, we only need to provide new values for periods, execution times and deadlines. Furthermore, we can toggle the keywords in Template 3.1 such as periodicity, preemptiveness, priority type, etc. While providing fresh values for the paremeters, we also need to consider the constraints between the keywords and parameters of the template. Consider the following two examples of constraints:

1. For rate-monotonic scheduling, the problem should have periodic tasks. However, for earliest deadline first, one can have sporadic tasks as well.
2. To create new problems based on Example 3.1, we can generate random values for p_1, p_2 and e_1 such that the following constraints hold:

$$\boxed{p_1 < p_2, \quad e_1 < p_1, \quad p_1 \nmid p_2}$$

where $p_1 \nmid p_2$ denotes "p_1 does not divide p_2".

Table 4 shows 10 of these randomly generated values with respect to the above constraints. Also just to put a bound on time, we added an additional restriction that $e_1, p_1, p_2 \in [1, 20]$. Each triple of values in Table 4 yields a new problem variant: for each of these, we can still ask for maximum value of e_2 such that there exists a feasible rate monotonic schedule.

4. CONCLUSIONS

In this paper, we discussed *automated exercise generation*, one of the technical challenges associated with the advent of massively open online courses (MOOCs). We presented an approach we are developing for the undergraduate embedded systems course at UC Berkeley. This approach is based on generalizing existing exercise problems into templates that

e_1	p_1	p_2	e_1	p_1	p_2
1	10	11	5	19	20
10	19	20	6	7	11
1	5	14	6	15	20
2	11	19	3	6	9
4	10	17	4	6	17

Table 4: Random Generated Parameters for RM Schedules

capture common structure, and adapting techniques developed for formal verification and synthesis to generate solutions and problems. We demonstrated our approach for two types of problems: model-based problems, and real-time scheduling problems.

This effort is very much a work-in-progress, and much remains to be done. Several problems in Lee and Seshia [14] do not fit the templates we presented in this paper, such as those that are based on code and involve reasoning about the effect of interrupts or threads. However, even these problems have their own templates. Indeed, we often train students to "find the template" by solving several problems and reflecting on the common solution technique that they applied to these problems.

A second, very interesting direction is to automatically generate customized feedback for students who make errors in their solutions. For model-based problems, this essentially reduces to the problem of error localization — a problem that is also well-studied by the formal methods and design automation communities.

We are starting to look into integrating the techniques described in this paper with model-based design frameworks with graphical modeling languages such as Ptolemy II [8], LabVIEW, and Simulink/Stateflow.

Finally, the ultimate goal would be an interactive exercise creation toolkit for instructors, where instructors express their creative insights about a particular problem (e.g., by stating the atomic propositions that define the new context for a model-based design problem), and the tool then generates a list of suggested new problems for use in the course. As a concrete example, suppose that we have a traffic light controller design problem for cars, and the instructor wants to turn this into a railway crossing problem. What is the best interaction model between instructor, tool, and students? What underlying computational engines do we need? How can students and instructors cooperate, assisted by algorithms, to provide customized feedback and a better learning experience? This paper is a first step towards developing such a toolkit, which we hope to make available to the wider community.

Acknowledgments

This work was supported in part by NSF CAREER grant #0644436, NSF Expeditions grant #1139138, and by an Alfred P. Sloan fellowship.

5. REFERENCES

[1] Coursera. `http://www.coursera.org`.

[2] edX. `http://www.edx.org`.

[3] Udacity. `http://www.udacity.com`.

[4] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[5] G. C. Buttazzo. *Hard Real-Time Computing Systems-Predictable Scheduling Algorithms and Applications*, volume 24. Springer-Verlag, 2011.

[6] T. Chea. Elite colleges transform online higher education. `http://finance.yahoo.com/news/elite-colleges-transform-online-higher-124855202.html`, August 2012. Associated Press.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[8] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[9] R. M. Hierons and M. G. Merayo. Mutation testing from probabilistic finite state machines. *Journal of Systems and Software (JSS)*, 82(11):1804–1818, 2009.

[10] S. Jha, R. Limaye, and S. Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 668–674, 2009.

[11] N. Jurkovic. Diagnosing and correcting students' misconceptions in an educational computer algebra system. In *In Symbolic and Algebraic Computation (ISSAC)*, pages 195–200, 2001.

[12] E. Lee and S. Seshia. EECS 149: Introduction to Embedded Systems. `http://chess.eecs.berkeley.edu/eecs149`.

[13] E. A. Lee and S. A. Seshia. An introductory textbook on cyber-physical systems. In *Proc. Workshop on Embedded Systems Education (WESE)*, October 2010.

[14] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. http://leeseshia.org, first edition, 2011.

[15] E. A. Lee, S. A. Seshia, and J. C. Jensen. Teaching embedded systems the berkeley way. In *Proc. Workshop on Embedded Systems Education (WESE)*, October 2012.

[16] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *ACM*, 20(1):46–61, 1973.

[17] S. Malik and L. Zhang. Boolean satisfiability: From theoretical hardness to practical success. *Communications of the ACM (CACM)*, 52(8):76–82, 2009.

[18] R. Singh, S. Gulwani, and S. Rajamani. Automatically generating algebra problems. In *Intl. Conf. of the Association for the Advancement of Artificial Intelligence (AAAI)*, 2012.