Formal Methods for Engineering Education

Sanjit A. Seshia, Ed.

June 30, 2015

Formal Methods for Engineering Education

Sanjit A. Seshia

Abstract

The advent of massive open online courses (MOOCs) has placed a renewed focus on the development and use of computational aids for teaching and learning. In Spring 2014, a graduate course was taught at UC Berkeley to explore the use of formal methods for a range of activities in online and large-scale education, including utomatic grading, synthesizing new problems, automatically solving problems, and creating and managing virtual laboratory environments. Students performed a range of projects exploring these topics for courses in computer science, electrical engineering, and other disciplines. This technical report includes the final reports on student projects along with a summary of some of the main lessons learned through in-class discussions.

1 Introduction and Motivation

The advent of *massive open online courses* (MOOCs) [16] promises to bring world-class education to anyone with Internet access. More significantly, the arrival of MOOCs have placed a renewed focus on the development and use of computational aids for teaching and learning. Over the last few years, several projects have been started to devise such computational aids based on techniques and tools developed in a range of areas in computer science and engineering including artificial intelligence, machine learning, programming languages, software engineering, formal languages and automata theory, human-computer interaction, visualization, and more.

In the Spring 2014 semester, an advanced graduate course was offered at UC Berkeley to explore the use of *formal methods* for education, particularly in various fields in engineering. The field of *formal methods* is concerned with the rigorous mathematical specification, design, and verification of systems [19, 6]. At its core, formal methods is about *computational proof techniques*: formulating specifications that form proof obligations, designing systems to meet those obligations, and verifying, via algorithmic proof search, that the systems indeed meet their specifications. The field has made enormous strides over the past few decades. Verification techniques such as model checking [4, 17, 5] and theorem proving (see, e.g. [15, 11, 8]) are used routinely in the computer-aided design of integrated circuits and have been widely applied to find bugs in software, analyze embedded systems, and find security vulnerabilities. At the heart of these advances are computational proof engines such as Boolean satisfiability solvers [14], Boolean reasoning and manipulation routines based on Binary Decision Diagrams (BDDs) [3], and satisfiability modulo theories (SMT) solvers [2].

MOOCs present a range of problems to which the field of formal methods has much to contribute. In particular, the following applications of formal methods in education have been identified:

1. *Automatic grading and feedback:* In automatic grading, a computer program verifies that a candidate solution provided by a student is "correct", i.e., that it meets certain instructor-specified criteria (the specification). In addition, and particularly when the solution is incorrect, the automatic grader (henceforth, *auto-grader*) should provide feedback to the student as to where he/she went wrong.

Several techniques from formal methods are applicable including systematic test generation, formal verification, equivalence checking, debugging techniques to provide error localization and feedback, methods to assign partial or extra credit, etc.

2. *Automated exercise generation:* Automatic exercise generation is the process of synthesizing problems (with associated solutions) that test students' understanding of course material, often starting from instructor-provided sample problems. Topics include the interface between natural language and informal descriptions and formal problem specifications, algorithms for problem generation with a range of objectives (testing various concepts, allowing self-paced learning, detering cheating, etc.), and solution generation, which involves solving the underlying decision problems in a manner that generates results that provide useful feedback to students.

3. *Design of virtual laboratories:* Several on-campus courses in science and engineering, such as in circuits, robotics, mechatronics, chemistry, etc. involve the use of a physical lab space. For such courses involving laboratory assignments, currently the best approximation that scales to hundreds or thousands of students is a simulated virtual laboratory. The goal is to provide the remote student with a learning experience similar to that provided in a real, on-campus lab, so that the student can gain enough expertise to transition without too much effort to the real lab setting. Formal methods can contribute much to the systematic design of the virtual lab assignments, automatic grading, and personalization to the needs of individual students and instructors. Some technologies can also be re-used in physical lab settings.

4. *Pedagogy:* There is a broad literature in the field of education on the use of cognitive models of learning to improve teaching and learning processes. Formal methods can contribute to this area by providing rigorous mathematical formalisms to capture these models along with algorithmic techniques to use these models along with technologies discussed in the items above to better personalize them for individual students and class settings.

The Spring 2014 UC Berkeley course, *CS294-98: Formal Methods for Engineering Education* [18] was a project-based course with the following structure. In the first month, foundational topics in formal methods, such as the use of mathematical logic and automata theory for formal modeling, and algorithmic techniques for verification and synthesis such as model checking, Boolean satisfiability (SAT) solving, and satisfiability modulo theories (SMT) solving were covered. Unlike classes on formal methods that teach students to conduct research in formal methods, the focus in this course was instead on how to effectively use and adapt formal methods for problems in (engineering) education. After the first month, class time was used to survey the emerging literature on the use of formal methods for education, and to focus on class projects. Relevant topics related to user interface design, machine learning, and evaluation studies were also covered via guest lectures and paper discussions.

This technical report includes the final reports on student projects along with a summary of some of the main lessons learned through in-class discussions. In this introductory paper, we provide a compendium of the main lessons learned by the instructor, students, and course support staff, so that this may serve as a platform for the broader community to build upon. The remainder of the report includes all final project reports submitted by graduate students enrolled in the course.

2 Questions Discussed

Throughout the course, and particularly in the final lecture, the instructor posed questions that cut across specific fields of study to identify some of the ways in which the topic "Formal Methods for Engineering Education" is distinct from other topics that have been taught and studied. In this section, we summarize these questions and the in-class discussion on them.

2.1 Difference between Automatic Grading and Formal Verification

How does the problem of automatic grading differ from other, more traditional, "industrial" applications of formal verification?

The following distinguishing features of the education domain emerged from the discussion:

- *Smaller problem scale:* The size of problems is much smaller for education than for industrial applications.
- *Greater solution diversity:* For a fixed problem specification, one needs to handle a much greater diversity of solutions in the education domain than in typical domains where the user of formal methods may be an expert engineer.
- *Availability of reference solutions:* In education, one typically has access to an instructor-provided "model" solution, whereas in industrial practice, one must typically make do with partial specifications of desired behavior.
- *Limited need for exhaustive verification:* In education, one may not need the solution to be correct for the entire space of inputs, but this is typically needed for industrial applications.
- *Need for quick feedback:* In education, it is essential for students to receive quick and timely feedback, e.g., on the order of a few minutes, so as to help them iterate quickly to learn a concept. In contrast, for industrial uses of formal methods, it may be acceptable to wait much longer to receive debugging information.
- *Tunable feedback:* In education, one may want to give different degrees of feedback to improve the learning experience — for instance, only reveal a high-level hint to the student to let them discover the flaw in the solution themselves, rather than revealing detailed line number-level information about the bugs in their code. In contrast, in industrial practice, one seeks to give the engineer as much debugging information as they can receive to quickly isolate and fix bugs.

2.2 Machine Learning vs. Formal Methods: Differences and Synergies

Several techniques proposed in the literature and deployed in MOOCs have been based on machine learning. Some of this literature was surveyed in the course. In this regard, two questions were posed:

(a) *What are the pros and cons of using machine learning vs. formal methods for automatic grading?*

The following advantages of formal methods were identified:

- Formal methods can give more detailed feedback (e.g. counterexamples).
- Formal methods can provide stronger theoretical guarantees about grading performance.

  - Formal methods can be effective where there is a precise problem specification.

  The following disadvantages of formal methods were listed:

  - Machine learning, being data-driven, can identify correlations in student data not easily represented in a formal model.
  - Machine learning methods can be more scalable than formal methods.
  - Machine learning can be more effective with ambiguous problem specifications, e.g., in a design problem.

  In general, there was agreement that there was much to be gained from combining techniques from the two areas. This led to the second question discussed below.

(b) *How can machine learning and formal methods be combined for automatic grading?*

  The following ideas emerged from the discussion:

  - Formal methods typically relies on error models; machine learning can be used to learn these from student data.
  - Machine learning can be used to mine (likely) formal specifications from a corpus of student solutions, which can be then used for further assessment and feedback.
  - Machine learning can be used to infer "cost models" to assign partial or extra credit on student assignments.
  - Machine learning can be used to find correlations between formally encoded course concepts and student errors, which can then be used to personalize the feedback given to students.
  - Formal methods can be used to generate test cases or examples (e.g., faulty solutions) to make machine learning algorithms more robust to missing data.

  In general, this combination of machine learning and formal methods appears to be a rich domain for developing new technologies for education.

2.3 Exercise Generation vs. Program Synthesis

In recent years, several algorithmic techniques have emerged to synthesize programs from multi-modal specifications and with formal guarantees of correctness. In this regard, the question was posed:

What are the key distinguishing characteristics between the problem of automatic exercise generation and other applications of program synthesis?

The following unique features of exercise generation were identified:

- *Large numbers of variants:* In education, one typically seeks to generate many variants of a single exercise. In traditional program synthesis, one needs only a single solution.
- *Varying difficulty levels:* In education, one needs exercises of varying levels of difficulty, whereas in program synthesis, one typically optimizes for a specific objective.
- *Pedagogical goal:* In education, the goal for synthesis is defined by a particular target concept the instructor wants to get across. In other words, the output of synthesis is just a means to some other end. In traditional program synthesis, the output of synthesis is the end.

- *Reference implementation:* In education, one typically starts with a sample problem that an instructor generates by hand, and then the synthesis task is to create variants of this problem. In other domains, such a sample problem may not be available.
- *Informal specification:* In education, the standard practice is to describe problems in natural language using informal notation used in class (e.g. diagrams or pseudo-code). In standard programs synthesis, problems may be defined more formally.

These differences must be kept in mind while adapting techniques from the program synthesis literature for the domain of education.

2.4 Virtual Lab vs. Physical Lab

What features would make the learning experience in a virtual laboratory a good approximation of that received in a physical lab?

The following points emerged from the discussion:

- *Realistic Simulator:* The simulator must capture all the aspects of the physical lab setting relevant to the concepts being tested in the lab assignment. This would include, for example, noise/fault models, resource constraints, and the programming experience.
- *Programmable Labs:* The virtual lab environment must offer instructors a way to customize the design of the virtual lab to their own course topics and learning objectives. The only way to achieve this is to make the lab environment highly programmable by instructors with an intuitive programming interface.
- *High-quality automatic grading and feedback:* One of the limitations of simulators is that they are only as effective as the input test stimuli used with them. It has been observed that without effective test cases, students will do just enough to pass the tests and this may not help them make their solutions robust enough for deployment in the physical world. Thus, high-quality testing and verification methods are essential to make sure that student solutions are "pushed" to cover all corner cases.

2.5 Bridging Natural Language and Formal Language

One of the major challenges in the field of formal methods is that while tools need formal specifications to operate effectively, engineers in industry rarely write formal specifications. This problem cannot be avoided in the education domain either. Thus, the question was posed: *What are some ways to address the natural language to formal language divide in education?*

The following points emerged from discussion:

- It is important to get the formal language to natural language translation correct for effective feedback, otherwise students may quickly "tune away" from the tool.
- In classes, where learning formal specification languages is an objective, one can use a "specification debugging" approach where a student attempts a translation between natural language and formal language, and the system returns "ramifications" of that statement. Similarly, students can be given problem statements in both natural language and formal language and asked to point out differences.

- Crowdsourcing, augmented by reputation systems, can be an effective way to generate high-quality translations from natural language to formal language, especially for specific domains from a trained body of students supervised by instructors and teaching assistants.

3 Course Projects

The course included eight research-oriented projects by graduate students as well as two implementation-oriented projects by undergraduates. The graduate student projects are listed below, sorted by the last name of the (first) author:

1. *Autograding and Problem Generation for Hybrid Automata Constructions- CybOrg*, by Nikunj Bajaj.
2. *SILVA: Student-in-the-Loop Verification of Assembly*, by Nicholas Carlini, Michael McCoyd, and Rohit Sinha.
3. *Online Monitoring of Signal Temporal Logic for Virtual Lab Autograding*, by Shromona Ghosh.
4. *Auto-Grading Dynamic Programming Language Assignments*, by Liang Gong.
5. *Autograding and Exercise Generation for Visual Finite State Machines*, by Zachary MacHardy and Jonathan McKinsey.
6. *Assistant for Learning Introductory Python Programs*, by Phitchaya Mangpo Phothilimthana and Cuong Nguyen.
7. *Computational Tools for Music Theory Pedagogy*, by Chris Shaver.
8. *Problem Generation for DFA Construction*, by Alexander Weinert.

4 Conclusion

The twin challenges of large-scale education (on-campus and online) and personalized education have thrown up several exciting research directions for the field of formal methods. This report summarizes the explorations on this topic performed during a Spring 2014 graduate course at UC Berkeley. In the year since the offering of the course, some of the projects have had a real impact on the practice of education. One such contribution is *CPSGrader*, which combines virtual lab software with automatic grading and feedback for courses in the areas of cyber-physical systems and robotics [10, 9, 7]. In particular, CPSGrader has been successfully used both on campus in *Introduction to Embedded Systems* at UC Berkeley [12] during Fall 2014 and in its online counterpart on edX [13] during the summer of 2014. Another such project is AutomataTutor [1], an online tutoring system for formal languages and automata theory that has been used in several courses around the world. We believe the area of formal methods for education has rich potential for impactful work, and we encourage the reader to contribute to this emerging area.

5 Acknowledgments

The material in this report has been developed jointly by the instructor, Sanjit A. Seshia, along with course staff (Alexandre Donzé and Garvit Juniwal), and all the students enrolled in the class. We also gratefully acknowledge the contributions of the guest lecturers: Derrick Coetzee, Jeff Jensen, Shalini Ghosh, Sumit Gulwani, and Wenchao Li.

References

[1] Automata Tutor. `http://www.automatatutor.com`.

[2] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[3] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[5] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[6] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[7] Alexandre Donzé, Garvit Juniwal, Jeff C. Jensen, and Sanjit A. Seshia. CPSGrader website. `http://www.cpsgrader.org`.

[8] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[9] Garvit Juniwal. CPSGrader: Auto-grading and feedback generation for cyber-physical systems education. Master's thesis, EECS Department, University of California, Berkeley, Dec 2014.

[10] Garvit Juniwal, Alexandre Donzé, Jeff C. Jensen, and Sanjit A. Seshia. CPSGrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory. In *Proceedings of the 14th International Conference on Embedded Software (EMSOFT)*, October 2014.

[11] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[12] Edward A. Lee and Sanjit A. Seshia. EECS 149 course website. `http://chess.eecs.berkeley.edu/eecs149`.

[13] Edward A. Lee, Sanjit A. Seshia, and Jeff C. Jensen. EECS149.1x Course Website on edX. `https://www.edx.org/course/uc-berkeleyx/uc-berkeleyx-eecs149-1x-cyber-physical-1629`.

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

[15] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[16] Laura Pappano. The Year of the MOOC. `http://www.nytimes.com/2012/11/04/education/edlife/massive-open-online-courses-are-multiplying-at-a-rapid-pace html`, November 2012.

[17] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, number 137 in LNCS, pages 337–351, 1982.

[18] Sanjit A. Seshia. CS294-98: Formal Methods in Engineering Education. `http://www.eecs.berkeley.edu/~sseshia/fmee/`.

[19] Jeannette M Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.

Autograding and Problem Generation for Hybrid Automata Constructions- *CybOrg*

Nikunj Bajaj
Instructor: Prof. Sanjit Seshia

May 16, 2014

Abstract

Grading and problem generation have been a long standing concern for instructors and advent of Massive Online Open Courses(MOOC) aggravates the situation . This new teaching paradigm escalates the student to teacher ratio by orders of magnitude making it practically impossible for instructors to provide personalized feedback to students. Generating new problem sets for practice and preventing classroom plagiarism adds another level of complexity. Therefore, a big challenge in MOOC is automating grading and problem generation. However, automation becomes extremely difficult when the problem involves continuous dynamics and worse if its hybrid. In this project, we present a step towards solving the problem of autograding and problem generation for questions based on hybrid systems. Specifically, we focus on a standard design problem that asks students to construct a hybrid automata, given certain properties that the system must satisfy. For auto-grading, we use a reference implementation of the problem and present a novel method for providing meaningful feedback even when no error model is available. We then demonstrate the use of error models to make the feedback more precise. For problem generation, we use a parameter based template and possible range of each of the parameters provided by the instructor. Our tool *CybOrg* works in loop with the SpaceEx platform which performs reachability and safety verification of continuous and hybrid systems.

1 Introduction and Motivation

Automation of grading and problem generation are indespensable in the new teaching paradigm of MOOCs with class size over tens of thousands and number of assignments of the order of hundreds of thousands [6] per course. In this context, a very important class of problem that tests student's depth of understanding of the problem and ability to apply the theoretical knowledge is design of systems meeting a set of given requirements. Alur et al address the problem of autograding of such design problems in [4]. However, they consider Deterministic Finite Automaton and discrete automata in general are not rich enough to represent the systems that involves physics.

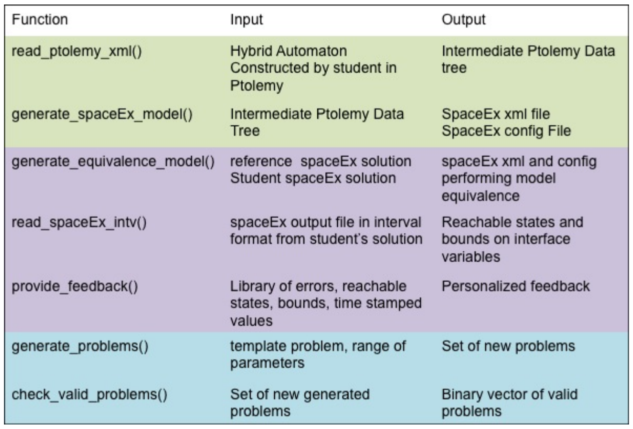
A strong and expressive formal object that can represent systems of a wide range is hybrid automaton. As defined by Rajeev Alur in [3] - *A hybrid automaton is an extended finite state machine whose state consists of a mode that ranges over finitely many discrete values and a set of real valued variables. Each mode is annotated with constraints that specify the continuous evolution of the system, and edges between modes are annotated with guards and updates that specify discrete transitions.* It is clear that an attempt for autograding of hybrid automaton constructions would entail formal verification and model checking. And, it has been pointed out by Becker et al in [5] that the theory of formal verification and model checking for hybrid sytems and the tools in this domain are not mature, making the problem further challenging. However, an interesting observation in this context is presented in [2] that claims that the problems frequently encountered in education domain are fairly simple and are restricted in system size. Therefore, a significant contribution can be made in the MOOC domain even with certain restrictions on the size of system. Specifically, for this project we use SpaceEx [1] that performs reachability and safety verification of continuous and hybrid systems where dynamics and transition guards are restricted to be affine.

Another major challenge in autograding is providing personalised feedback to students. In this project we present a novel technique of providing feedback even in the absence of error models. We also demonstrate the use of error model, an idea inspired from Singh et al's work [10], to make the feedback more precise. For problem generation we use a template based approach as presented by Sadigh et al in [9]. The rest of the report is organized as follows. We define the goals of the project in section 2 followed by the system overview in section 4. We also present the technical details of conversion of models from Ptolemy to SpaceEx in section 5, approach with some examples of autograding in section 6 and approach with an example of problem generation in section 7. Finally we present the results in section 8 and conclusions and future work in 9.

2 Problem Definition

In this project we develop a tool- *CybOrg* for autograding and problem generation of hybrid automaton constructions. Specific functionalities of the tool can be summarized as-

- **Ptolemy to SpaceEx conversion**: Given a model constructed in Ptolemy(which is a widely used software), automatically convert it to a SpaceEx model(which is the tool used for safety verification and model equivalence). This provides a student friendly front end for construction of hybrid automaton.
- **Autograding of hybrid automaton constructions**: Given a student solution, a reference solution provided by the instructor and possibly a library of error models- provide precise and personalized feedback to students on the correctness of their model and help them to localize mistakes, if any.
- **Automatically generating similar design problems**: Given the solution of a sample problem by the instructor and a template of the problem in abstracted form, generate a set of new problems of similar difficulty level.



Function	Input	Output
read_ptolemy_xml()	Hybrid Automaton Constructed by student in Ptolemy	Intermediate Ptolemy Data tree
generate_spaceEx_model()	Intermediate Ptolemy Data Tree	SpaceEx xml file SpaceEx config File
generate_equivalence_model()	reference spaceEx solution Student spaceEx solution	spaceEx xml and config performing model equivalence
read_spaceEx_intv()	spaceEx output file in interval format from student's solution	Reachable states and bounds on interface variables
provide_feedback()	Library of errors, reachable states, bounds, time stamped values	Personalized feedback
generate_problems()	template problem, range of parameters	Set of new problems
check_valid_problems()	Set of new generated problems	Binary vector of valid problems

Figure 1: Functions supported by CybOrg

3 Challenges

The idea of autograding requires checking the hybrid automaton constructed by student against certain specifications which entails verification of safety properties. We use SpaceEx for this purpose. In this process of using SpaceEx in loop with our tool *CybOrg* we experienced many research and implementation challenges. First of all, SpaceEx is not a frequently used tool in the student community and it is unreasonable from a MOOC perspective to expect every student to learn the semantics of SpaceEx for constructing models. This entailed development of a nice front end for students and doing the conversion to SpaceEx models at the backend by the tool itself. Secondly, SpaceEx does not accept safety properties in temporal Logic format and it just accepts hybrid automaton. So specifying the safety properties was not straightforward. We use a reference solution provided by the instructor for this purpose(details follow in section 6). How to make the tool insusceptible to structural differences of the model was also a challenge. This problem was addressed by treating the model as blackbox and verifying only the interface variables. How to use the SpaceEx output, which does not provide specific error traces, to provide meaningful feedback to students in case of incorrect solutions was far from trivial. Generating new problems and ensuring that they make physical sense using only reachability properties turned out to be very challenging(discussed in section 7).

4 Approach

In this project we provide a step towards solving all three problems mentioned in section 2 by developing the tool *CybOrg*. It derives its name from the blend of the terms (Cyb)ernetics and (Org)anism. Traditionally, the term is used for an organism whose capability is enhanced by the aid of technology [8]. In this context, we use this tool with the instructor in loop(details follow in subsections below) and hence, the term is used for a computer program whose capabilities are enhanced by the aid of human intelligence. Also, in this project we specifically focus on hybrid systems which has a computational part and a physical part. The word cybernetics is a representative of the computational world and organism is the representative of the physical world. The functionalities of *CybOrg* can be studied with the flow diagram-

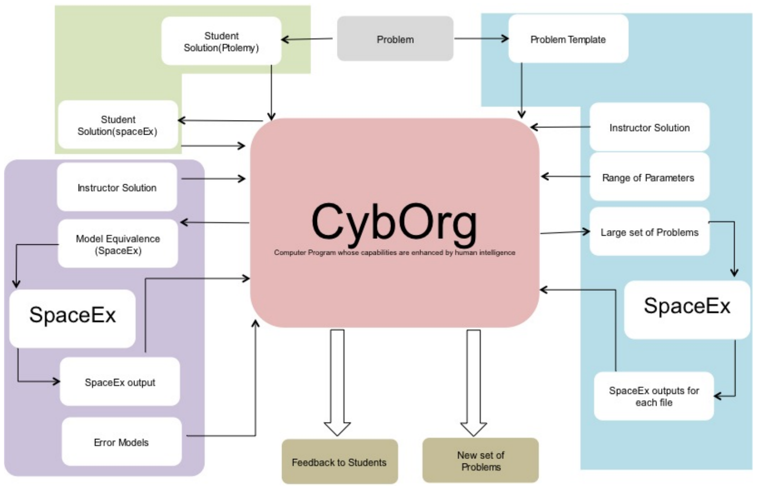


Figure 2: Input Output Interface of CybOrg

Three important functionalities of *CybOrg* are highlighted with 3 colors in the flow diagram- green for converting *Ptolemy models to SpaceEx*, purple for *autograding and feedback*, blue for *problem generation*.

The flow starts from the *problem* given to the student who constructs a hybrid automaton in Ptolemy corresponding to the problem. The Ptolemy model is given to *CybOrg* which translates it to SpaceEx model. *CybOrg* runs certain simulations of this model using SpaceEx to check for preliminary reachability properties. Then, it uses the SpaceEx version of student solution and instructor solution constructed in SpaceEx to generate a hierarchical SpaceEx model for equivalence checking. This model is fed to SpaceEx that checks if the traces of interface variables of the two models lie within a certain threshold. Using the output of simulation traces on student's solution and the model equivalence, *CybOrg* provides feedback to the students. It also takes in a library of error models to provide more precise feedback. For problem generation, *CybOrg* uses a template based method. For a given problem, instructor abstracts it to a generic template with flexibility to generate new problems by varying parameters of the template. *CybOrg* takes in the range of parameters provided by the instructor and generates new problems by sweeping each parameter across the range. Then a new SpaceEx model is generated corresponding to every problem which is sent to SpaceEx. The solution is verified for reachability properties and if possible for bounds on interface variables against the instructor's solution to the original problem. Based on the outputs of SpaceEx, *CybOrg* marks certain problems as valid.

5 Ptolemy to SpaceEx

Ptolemy and SpaceEx models are both represented as xml files but translating from one to other is more than merely tranforming structure of an xml file. Ptolemy is a simulation tool and SpaceEx is a formal verification tool, ptolemy deals with input output based causal models whereas SpaceEx supports acausal semantics and these factors result in challenges more fundamental than syntactic. This in turn also imposes certain restrictions on the models that *CybOrg* can handle. The details of translation and restrictions on the model can be best studied using an example. Here we consider the automata construction for the problem -*Design a thermostat for a room whose temperature is being monitored. If the temperature falls below* 18^o *C, turn the heater on increasing the temperature at the rate of* 2^o *C/ min and if the temperature rises above* 21^o *degrees Celsius, turn it off decreasing the temperature at the rate of* 1^o *C/min. Consider, initially the temperature is* 20^o *C and the heater is turned off.*

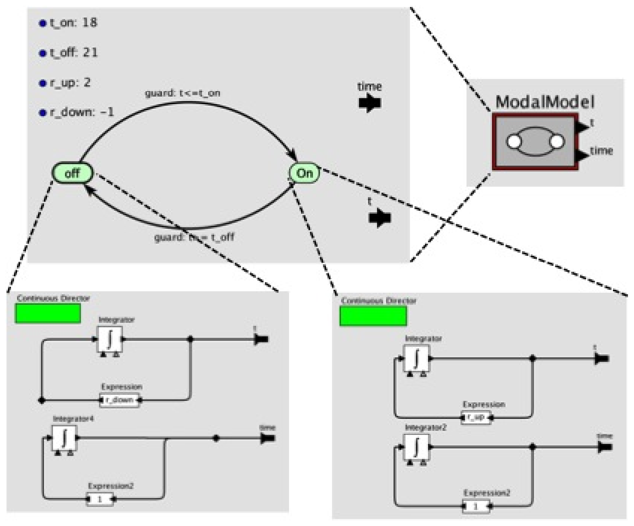


Figure 3: Ptolemy Implementation of Thermostat

The Ptolemy model implementing the themostat is shown in the figure 3. We see that a hybrid automaton in Ptolemy is implemented in a hierarhical manner where the state machine is a refinement of the *ModalModel* actor and the dynamics are also modeled as refinement of the states using i/o based actors.

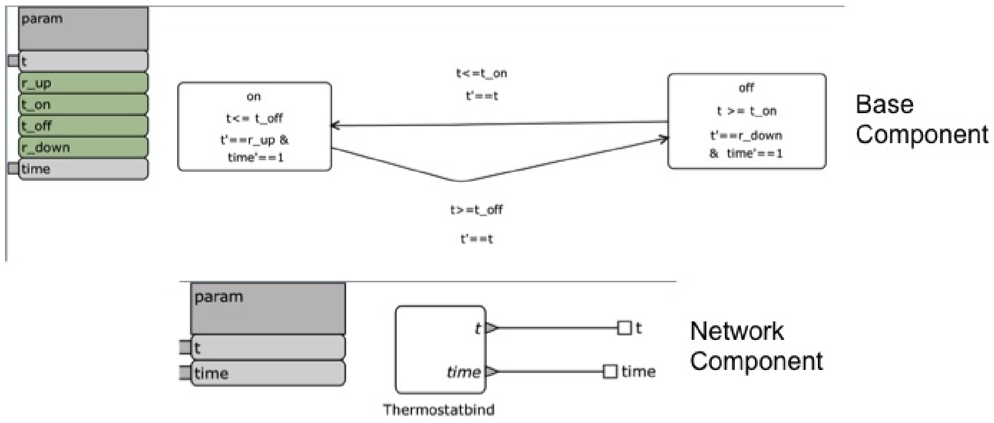


Figure 4: SpaceEx Implementation of Thermostat

SpaceEx implementation of same Thermostat is shown in figure 4. In SpaceEx the automaton is built as a *Base Component* which is then instantiated by a *Network Component* by supplying all the paramaters.

Clearly, on a high level the two softwares differ heavily in their implementation. Some of the low level differences and challenges assosciated with them are discussed below-

- **Syntatic Difference**- In Ptolemy the initial state of the machine and initial condition of the variables are specified in the model itself whereas in SpaceEx all the initial conditions are provided in a separate configure file to the software. In Ptolemy xml files the transitions are modeled as *relations* contain the set action, output actions and guards but the source and destination information of the transitions are modeled as a separate entity called *links*. In SpaceEx xml all information related to a transition is provided under one entity called *transition*. This poses some implementation challenge of fetching information from two entities by name tag matching and putting everything together in one data structure.

- **Semantic Difference**- Ptolemy does not require any explicit declaration of invariants in state whereas SpaceEx does. So in translating models from Ptolemy to SpaceEx, *CybOrg* has to generate this information. This is done by negating the transition guards. For instance, in *off* state of Ptolemy implementation transition guard is $t \leq t_{on}$ and negating the guard, $t > t_{on}$ invariant of the *off* state can be easily obtained. If there are multiple transitions from one state, then one solution could be taking the union of all transition guards and negate that to obtain the invariant. However, SpaceEx restricts the invariants to be conjunction of inequalities. For instance, if two transitions from a state have guards $a \geq 0$ and $b \geq 0$ then in no way can we represent $\sim (a \geq 0 \vee b \geq 0)$ as a conjunction of inequalities (negation operator is also not supported by SpaceEx invariants). This restricts the models to have only transition per state.

- **Structural Difference**- In Ptolemy, the dynamics are modeled using a network of actors and there can be multiple ways of modeling a differential equation, whereas SpaceEx directly supports dynamics in differential equation format. So in translating the model from Ptolemy to SpaceEx *CybOrg* has to obtain the differential equation from the network of actors. In this context, its important to notice that

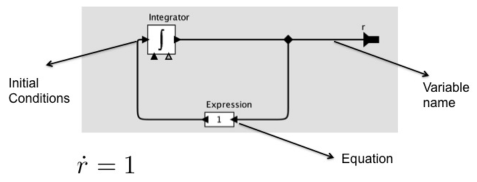


Figure 5: Obtaining dynamics of variables from Ptolemy

any differential equation of the form $\dot{x} = Ax + Bu$ (SpaceEx supports only affine dynamics) can be represented by the 3 actors as shown in figure above. This does not restrict the complexity of models *CybOrg* can handle but it does restrict the way students should model dynamics in their solution.

6 Autograding

For autograding its very important to define the notion of erroneous model. In case of hybrid automaton, continuous trace of the interface variables have to be verified over a period of time. In this context, we define Error- *If the difference between the output traces in the students and instructors' solution exceeds a certain threshold, we call the model erroneous.*
CybOrg provides feedback to students on correctness or efficiency of solution in multiple stages starting from preliminary feedback to more precise feedback. It uses error models only to help localize the errors better but can provide meaningful feedback even in the absence of error models. The feedback generation scheme used by *CybOrg* can be summarized as follows-

- **Preliminary Feedback** Preliminary information, like bounds on variables, unreachable states is obtained by simulating students solution on SpaceEx and reading its output in .intv format.
- **Precise Feedback** Precise feedback is given by finding the exact time when difference first crosses the threshold by running model equivalence(on student and instructor's solution) using SpaceEx and reading .gen output.
- **Error Model** If the feature vector of error characterstics is available in the library then student can be pin pointed to the error along with providing the generic feedback that may help to localize the errors.

6.1 Preliminary Feedback

SpaceEx performs preliminary checks on the model for reachability of states and bounds on interface variables and provides the output in .intv(read as interval format. An example is given in the appendix section 11). This itself can detect many syntactic errors made by the student. And this information can be easily used to localize errors efficiently.
For instance, consider the Ring problem - *Construct a timed automaton that provides the on and off signals as outputs, to be connected to the inputs of the tone generator. Your system should behave as follows. Upon receiving an input event ring, it should produce an 80 s-long sound consisting of three 20 s-long bursts of the pure tone separated by two 10 s intervals of silence.*

The student's solution is shown in figure 6. The machine starts from the state *wait* and upon receiving the input ring transitions to state *one* producing a sound for 20 seconds, then the state machine transitions to state *two* for 10 seconds where no sound is produced and similarly after producing an alternating on-off waveform for 80 seconds, goes back to wait state. The variable keeping track of time spent in a particular state is *r*

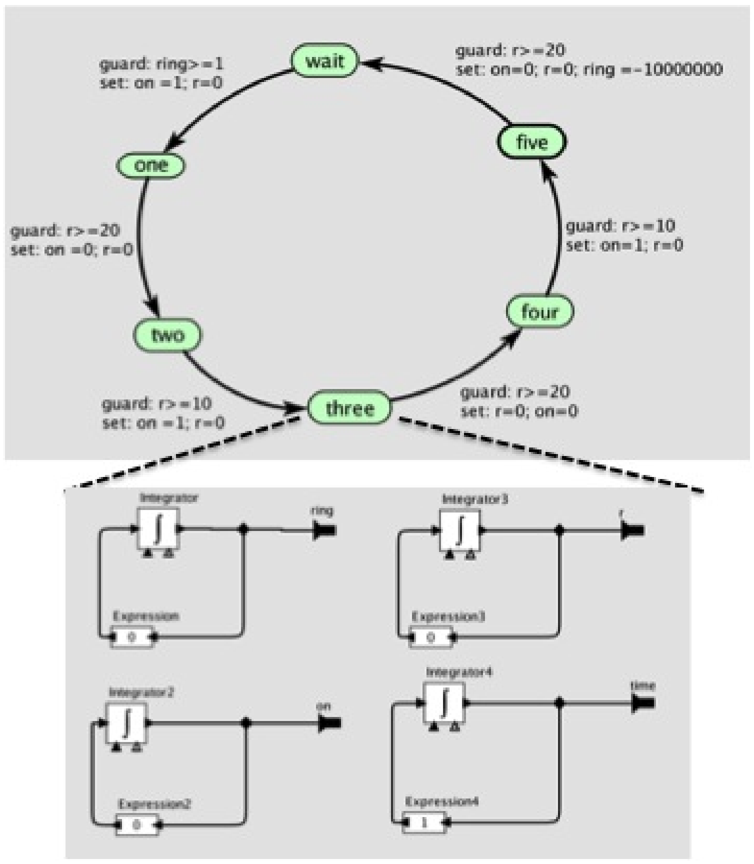
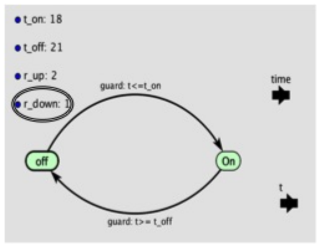


Figure 6: Wrong Dynamics of the problem

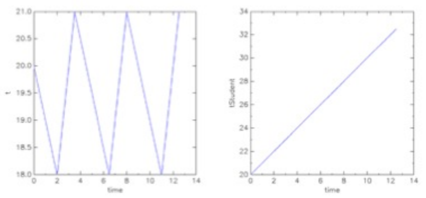
and it is reinitialized to zero at every transition and should increase at a rate of 1 in every state. Suppose for instance, in state *three* the student put the dynamics of r as $\dot{r} = 0$ instead of $\dot{r} = 1$. Clearly, *r* never increases beyond 0 and state machine never reaches states *four* and *five*. In this case an initial simulation of the model by SpaceEx can provide the information that state *four* and state *five* are not reachable and this information is indicative of a probable error in state *three* or the transition from *three* to *four*.

Similarly, in the thermostat example suppose the student forgot to put the negative sign in case of rate of decrease of temperature. Specifically, in the *off* state, when the heater is off the temperature should decrease with the rate of $1^oC/min$ making $r_{down} = -1$. However, student forgot the negative sign. In this case, the temperature keeps on increasing in the off state and the thermostat never goes to the on state. Preliminary simulations of the student solution gives two results - Temperature goes beyond bounds because it keeps on increasing beyond 21^o C and state *on* is never reachable.

Although, this method is good to identify errors that result in unreachable states or variables going out of bounds it cannot detect mistakes that result in none of the above. For instance, in the themostat problem if the student happens to swap the magnitude of rate of increase and decrease of temperatures in on and off state respectively i.e make $r_{up} = 1$ and $r_{down} = -2$, both on and off states are reachable and the temperature also stays within bounds of 18 and 21. Therefore, smarter technique is required to identify mistake and localize error. For this we use equivalence checking between student and instructor's solution.



(a) Rate Decrease Sign Change



(b) Left and right figures show expected and obtained temperature plots

Figure 7: Preliminary check obtained from SpaceEx simulation

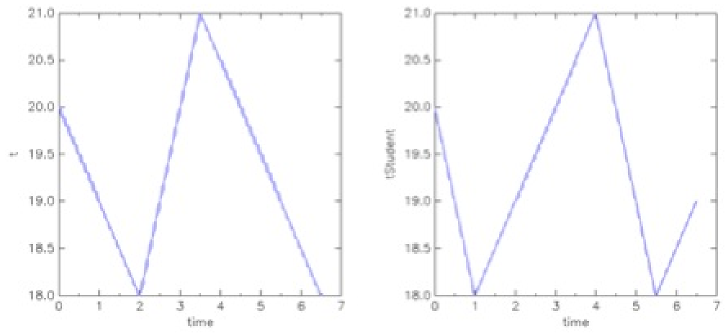


Figure 8: Left and right figures show expected and obtained temperature plots with error in rate of change of temperature

6.2 Generating Model Equivalence

In this section, I show how to generate SpaceEx model for equivalence checking using the two solutions. Let us consider figure 9. The model is constructed by generating a hierarchical network component of SpaceEx that subsumes both the base and network components of student and instructor's solution. It also requires linking new interface variables of the hierarhcical component to appropriate variables from the two solutions.

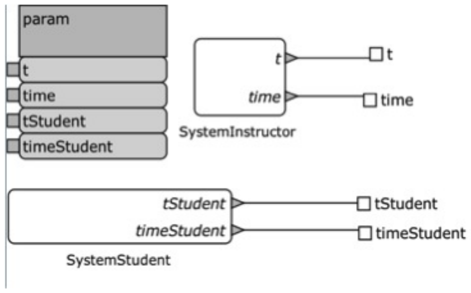


Figure 9: Model Equivalence file SpaceEx

Another aspect of constructing SpaceEx model is generating the configure(.cfg) file. A sample configure file is shown in appendix section 11. The initial States of both the state machines(student and instructor's), initial values of all interface variables is to be fetched from respective configure files and dumped in new configure file, output format of SpaceEx is to be specified(currently *CybOrg* deals with two formats - .intv and .gen) and the safety property is specified in terms of forbidden state. For instance, in case of thermostat example we say that the a forbidden state is reached if any of the interface variable of the student's solution differ from the instructor's by a threshold of 0.5.

6.3 Precise Feedback

In this section we present the most effective method of identifying error and providing feedback. Once we specify the safety property, in terms of forbidden state for the model equivalence file, SpaceEx output in .gen format provides exact time instants when the forbidden states are reached. For instance, the plots of rate magnitude error with forbidden states specified can be seen in figure 10. Through this we can get exact time instants when student's solution differ from that of instructor's and can be very helpful in localizing the error.

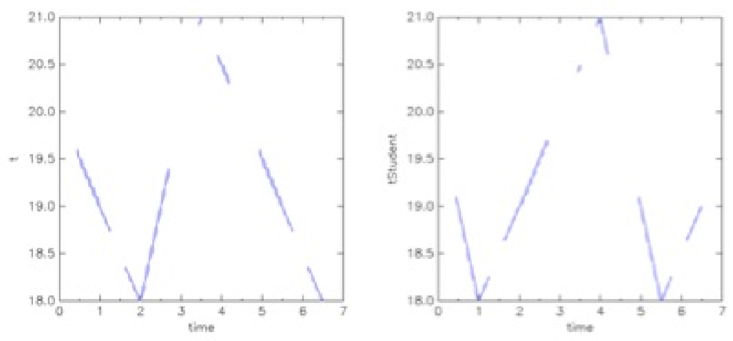


Figure 10: Plots of student and instructor solution in forbidden state

For instance, in this case the first time when the two solution differs is at $time = 0.5$. Similarly, let us consider again the error of putting wrong dynamics in state *three* in the Ring problem 6.1, as shown in figure 6 in the context of model equivalence. SpaceEx provides information that the outputs of the student and instrucor's solution first differ at time instant $t = 50$ secs. Using this information the student can easily localize that ideally the state machine should have been in state *three* at $time = 50$ and if the error occurs at time instant *50* then most likely the mistake is in the same location.

6.4 Error Models

In the above sections we presented a generic scheme of providing meaningful feedback to students even in the absence of any error models. Now, we demonstrate how the presence of an error model can make the feedback much precise because in a MOOC scenario, it has been observed that similar mistakes are observed in solutions of many students. Infact, as mentioned in [6], a programming assignment in a Machine Learning class offered as a MOOC that observed over 30,000 submissions had approximately 200 distinct solutions indicating the high repeatition rate of errors. Therefore, to provide precise feedback to students a library of error models can be very meaningful. For instance, we started developing a similar library for the Thermostat problem where a error committed by student is characterised by a set of violated properties detected by SpaceEx. Some examples are mentioned in table below

t	time	on	off	timeStamp	Feedback
0	0	0	1	0.5	Possible Error in Rate Decrease Expression
0	0	0	1	0.5	Possible Error in Dynamics
1	1	1	1	2	Possibly Rate Increase-Decrease Magnitude Swapped
1	1	1	1	0.05	Possible error in putting temperature bounds

The table is to be read as follows- Element of column 1 is 0 if variable *t* exceeds bounds and 1 if it remains withing bounds and similarly for column 2. Element of column 3 are 1 if state *on* if reachable else it is 0 and similarly for column 4. Element of column 5 represents the first time stamp when student's solution differ from that of instructor's. Column 6 represents possible mistakes in the solution that may correspond to the properties mentioned in the first 5 columns. Properties of these kinds are referred to as feature vectors.

In case of any erroeneous solution *CybOrg* constructs a similar feature vector based on the outputs of SpaceEx and if it finds a feature vector in the library matching to that of current erroneous solution it points the student to the plausible set of errors. If it doesn't find a matching feature vector then it provides generic feedback in the form of reachable states, variable bounds, time stamps when forbidden states is reachable and prompts the instructor to update the library. Clearly, different errors can have similar feature vectors but still such a library can help localizing the error much better. Moreover, since we use a machine learning technique for this and since the repeatation rate is pretty high, its reasonable to expect that the library will become exhaustive very soon and providing feedback can be very efficient.

7 Problem Generation

In automation of problem generation, specifically design problems, the biggest concern is whether a meaningful solution exists towards the new problem. In this project we provide a small step for solving this problem. Here, our focus is on a specific class of problems that can be represented in the form of an abstract template where certain parameters can be varied to generate new problems. The contribution of this project in the domain of Problem Generation is the effective use of SpaceEx to check the validity of the new problem and also eliminating problems that are not of same *diffuclty level*. In this context, we define two problems to be of the same difficulty levels if the corresponding solutions have similar reacability properties i.e. the solution can be obtained using same number of states. This definition makes sense in our case because we generate new problems only by varying the parameters and if there exists a solution to the new problem in less number of states then most likely some properties of the original problem are trivially satisfied. The overall *CybOrg* approach for generating new problems can be summarized as-

- Take in a sample problem, its template, instructor's solution to sample problem and a range of values for the parameters
- Sweep each parameter across the range and generate new problems
- Solve the new problem and verify that the new automaton satisfies the requirements(in terms of bounds) and has similar reachability properties (all states are reachable)
- If both conditions are satisfied, problem is marked as valid
- Instructor verifies in the final stage.

We demonstrate this using an example. Suppose we consider the problem taken from one of the assignments of EECS 219D course offered in Spring 2014 -
Temperature of a room is given by $w(t) = w(0) + \int_0^t (r_{up} z(\tau) - r_{down}(1 - z(\tau)))d\tau$ where r_{up} is constant representing the rate of heating when heater is on and r_{down} is the rate of cooling when heater is off and w(0) is the initial temperature of the room. In particular, suppose $C = 20$ secs, $w(0) = 22$ degrees, $r_{up} = 0.2$ degrees per second, $r_{down} = 0.1$ degrees per second, $t_{on} = 20$ degrees, $t_{off} = 24$ degrees. How often will the heater turn on and off. What are the maximum and minimum temperatures reached?
We first transform it to a template where certain parameter values are abstracted to be a variable. The template looks like
Temperature of a room is given by $w(t) = w(0) + \int_0^t (r_{up} z(\tau) - r_{down}(1 - z(\tau)))d\tau$ where r_{up} is constant representing the rate of heating when heater is on and r_{down} is the rate of cooling when heater is off and w(0) is the initial temperature of the room. In particular, suppose $C = 20secs$, $w(0) = p_1$ degrees, $r_{up} = p_2$ degrees per second, $r_{down} = p_3$ degrees per second, $t_{on} = p_4$ degrees, $t_{off} = p_5$ degrees. How often will the heater turn on and off. What are the maximum and minimum temperatures reached?
The instructor also provides the range of these 5 parameters as shown in the table below

Parameter Name	Minimum Value	Maximum Value
p_1	15	25
p_2	-3	7
p_3	-4	6
p_4	13	23
p_5	16	26

CybOrg uses these information to generate new problems by changing the parameter values of the original problem as per the table. For each new problem, it generates a new solution based on the instructor's solution to the original problem and simulates it using SpaceEx. Based on the preliminary checks on reachability of states and bounds (like the one used for feedback generation in the preliminary stage) it discards certain problems as invalid. Specific to the thermostat case *CybOrg* generated 50 new problems and discarded 14 amongst them. Upon manual inspection of the valid and discarded problems we see that the results make absolute sense. For instance, some of the reasons of discarding problems are mentioned below-

- Problem where rate of decrease becomes 0, making the new problem trivial. This case was deetected by unreachable state.
- Rate of temperature increase is negative, which does not make any physical sense. This was detected by temperature going beyond bounds
- t_{on} becomes greater than t_{off} and again the problem does not make any physical sense. This was also detected by unreachable state.
- Rate of increase of temperature become 0, which also makes the problem trivial. Detected by out of bound variable and unreachable state.

8 Results

The techniques discussed above were applied for autograding and problem generation for many problems in Chapter 4 (Modal Models) of the book [7]. Here we provide some selected feedback generated by *CybOrg* for distinct student solutions, for different problems discussed above

Problem	**Mistake**	**Feedback**
Thermostat [5]	Rate Decrease Sign Change[6.1]	Temperature goes beyond bounds. State on is not reachable. Temperature first differ at time 0.5 sec. Possible error in rate decrease expression or dynamics.
Thermostat [5]	Wrong Dynamics in state[11]	Temperature goes beyond bounds. State on is not reachable. Temperature first differ at time 0.5 sec. Possible error in rate decrease expression or dynamics.
Thermostat [5]	Rate Change Magnitude Swap[6.1]	Temperature differ from Instructor at multiple locations, like t=0.5, t=1.7, t=3.9. Possible error in magnitude of rate change.
Ring Problem[6.1]	Wrong Dynamics State 3[6.1]	States four and five are not reachable. Output differs after time t = 50 secs
Ring Problem[6.1]	Forgotten Set Action in 3^{rd} transition	Output differs after time t = 50 secs

Although we tabulate only the representative results the performance of the tool was verified on a broader range of problems and student solutions. We should also note that in case of Thermostat problem *CybOrg* is able to point to errors precisely because we developed a library of error models for the same. Still, the feedback provided for the Ring problem, although generic is very useful in efficiently localizing the mistakes.We should also note that since two of the mistakes in Thermostat problem resulted in the same feature vector of error model the feedback generated by the tool is same in both cases.

We also tested the performance of the tool for problem generation on a few design problems from the chapter that could be transformed in the form of a template and obtained very promising results. *CybOrg* generated 60 new problems for the Sequence Generator example(Problem 1 of Chapter 4 in [7]) out of which it marked 38 of them as valid. Similarly, we tried for the ring problem and it generated 70 new problems out of which 54 were marked as meaningful.

9 Conclusions and Future Work

In this project, we developed a tool *CybOrg* for autograding and problem generation for hybrid automata constructions. To my knowledge, this is the first work done in the MOOC domain that involves verification of hybrid systems. We also proposed a novel idea of providing meaningful feedback to students without any error models and using the error model, if available, only to help the students localize error more precisely. This project also illustrates efficient use of SpaceEx for generating new problems and providing feedback. However, the tool imposes some limitations on the complexity of the models. We think that there is some scope of extending the scope of *CybOrg* to more complex models inspite of the limitations of SpaceEx. Also, it will be interesting to explore machine learning techniques for problem generation. For instance, the instructor can train the tool by providing scores to the new generated problems which can be used in the future to generate more meaningul problems.

10 Acknowledgement

I thank Prof. Sanjit Seshia for introducing me to the new research area of applying Formal Methods in the Education domain, and his constant support and feedback on the ideas and implementation of the algorithms in the course of the project. I would like to convey my thanks to Alex Donze and Goran Frehse for helping me with the working of SpaceEx and special thanks to Alex Donze for providing much valuable insights on how can the tool be better used in context of the project. I would also like to thank Liangpeng Guo and Antonio Iannapollo for helping me with software related issues. Last but not the least, I would like to thank every student attending the class EECS 294 for their questions and feedback during the class presentations and interesting ideas that came up during the paper discussions.

References

[1] Spaceex. `http://spaceex.imag.fr/`.

[2] U. Z. Ahmed, S. Gulwani, and A. Karkare. Automatically generating problems and solutions for natural deduction. In *In IJCAI*, 2013.

[3] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, London, UK, UK, 1993. Springer-Verlag.

[4] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of dfa constructions. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI'13, pages 1976–1982. AAAI Press, 2013.

[5] B. Becker, M. Behle, F. Eisenbrand, M. Frnzle, M. Herbstritt, C. Herde, J. Hoffmann, D. Krning, and B. Nebel. *Bounded Model Checking and Inductive Verification of Hybrid Discrete-Continuous Systems*. Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2004.

[6] J. Huang, C. Piech, A. Nguyen, and L. Guibas. Syntactic and functional variability of a million code submissions in a machine learning mooc.

[7] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. Lee and Seshia, 1 edition, 2010.

[8] P. C. McGuire, J. Orm, E. D. Martnez, J. A. R. Manfredi, J. Gmez-Elvira, H. Ritter, M. Oesker, and J. Ontrup. The cyborg astrobiologist: First field experience. *CoRR*, 2004.

[9] D. Sadigh, S. A. Seshia, and M. Gupta. Automating exercise generation: A step towards meeting the MOOC challenge for embedded systems. In *Proc. Workshop on Embedded Systems Education (WESE)*, October 2012.

[10] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. *SIGPLAN Not.*, 48(6):15–26, June 2013.

11 Appendix

.intv Output of SpaceEx for Ring Problem

```
Computing reachable states... Iteration 0... 1 sym states passed, 1 waiting 0.231001s
Iteration 1... 2 sym states passed, 1 waiting 0.626001s
Iteration 2... 3 sym states passed, 1 waiting 1.04201s
Iteration 3... 4 sym states passed, 1 waiting 1.36101s
Iteration 4... 5 sym states passed, 1 waiting 0.872001s
Iteration 5... 6 sym states passed, 1 waiting 1.60701s
Iteration 6...
WARNING (incomplete output) Reached time horizon without exhausting all states, result is incomplete.
7 sym states passed, 0 waiting
Iteration 6 done after 1.71001s
Found fixpoint after 7 iterations.
Computing reachable states done after 7.45101s
Output of reachable states...
Bounds on the variables over the entire set:
on: [-0.00610803,1.02251]
ring: [-1e+07,1.01407]
r: [-0.021737,20.0001]
time: [3.46945e-17,120.017]

Location-wise bounds on the variables:
Location: loc(Ringbind)==three
on: [0.994199,1.02148]
ring: [0.978628,1.0136]
r: [-0.00108485,20.0001]
time: [30.0075,50.0145]

    Location: loc(Ringbind)==one
on: [0.994669,1.02037]
ring: [0.980819,1.01323]
r: [-0.000270754,20.0001]
time: [0.00814971,20.0134]

    Location: loc(Ringbind)==four
on: [-0.00610803,0.0219214]
ring: [0.977207,1.01382]
r: [-0.0013365,10]
time: [50.007,60.0153]

    Location: loc(Ringbind)==five
on: [0.993502,1.02251]
ring: [0.975473,1.01407]
r: [-0.00159347,20.0001]
```

```
time: [60.0064,80.0161]

    Location: loc(Ringbind)==two
on: [-0.00552226,0.0210495]
ring: [0.979826,1.01334]
r: [-0.000716402,10.0001]
time: [20.0079,30.014]

    Location: loc(Ringbind)==wait
on: [-0.00536415,0.013434]
ring: [-1e+07,1]
r: [-0.021737,0.0138579]
time: [3.46945e-17,120.017]

    4.14301s
ATTENTION: Warnings were issued. Warnings on incomplete output: 1
```

SpaceEx configure file Thermostat

```
system = ThermostatEquivalence
initially = ”tInstructor == 20 & timeInstructor == 0 & loc(ThermostatInstructorbind) == off
& t == 20 & time == 0 & loc(Thermostatbind) == off ”
scenario = ”supp”
directions = ”uni32”
sampling-time = 0.1
time-horizon = 40
iter-max = 20
output-variables = t, tInstructor, time, timeInstructor
output-format = ”INTV”
rel-err = 1e-12
abs-err = 1e-13
forbidden = t-tInstructor > 0.5 | tInstructor-t > 0.5
forbidden = time-timeInstructor > 0.5 | timeInstructor-time > 0.5
```

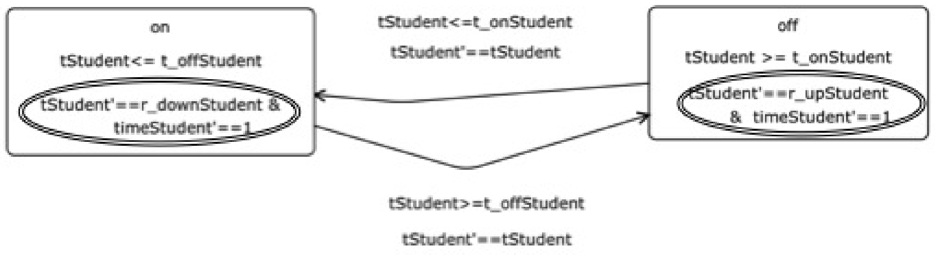


Figure 11: Wrong Dynamics in State in Thermostat problem

SILVA : Student-in-the-Loop Verification of Assembly

Nicholas Carlini, Michael McCoyd, Rohit Sinha

{npc, mmccoyd, rsinha}@cs.berkeley.edu

Abstract

We present a method of teaching students system code through formal verification. By asking students to write assertions and loop invariants, not only can they prove interesting properties about system code, they can also be forced to understand how that code works. Our analysis is preformed at the x86 binary level, and we argue this is an appropriate level for systems code. We verify x86 binary programs by using the BAP framework to convert it to a simpler IR. We then write a tool to convert this IR to Boogie, a frontend language for SMT solvers. With this we verify the student assertions and use their invariants to prove properties about the given program. We demonstrate verification of x86 binary is practical and efficient when done at small scale, as is typical of student assignments.

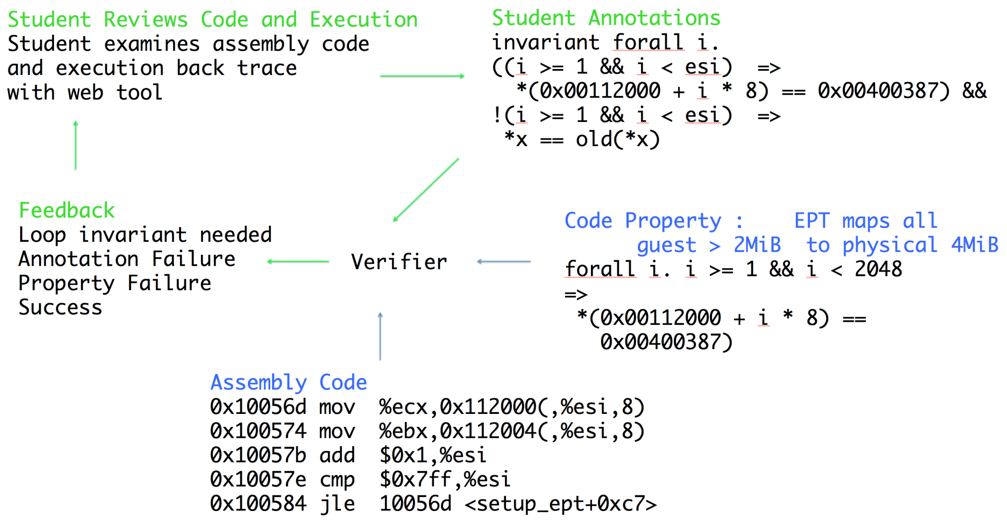
1 Introduction

Much low-level, security critical software is partially written in assembly — it is critical that the developer understands the semantics of each assembly instruction. We posit that it is important to teach students about the internals of x86 instruction set architecture. To that end, we present a methodology for teaching students about x86 binary programs. Determining program invariants is arguably an effective method of understanding a program's behavior. We propose a teaching method based on constructing interesting, albeit small, examples of assembly programs, and asking students to demonstrate their understanding by annotating invariants about each program.

The instructor creates an assignment consisting of an assembly program, along with a property that the instructor would like to be verified about the program. The student's task is to annotate the program with invariants that will guide the theorem prover in proving the program property. A successful submission consists of a set of annotations that allows the theorem prover to conclude that the program satisfies the property — we assume that the instructor provides correct programs. Furthermore, our tool provides feedback to students on their submissions. If the students provide an incorrect annotation, we present a counter-example trace demonstrating a program execution that violates the student's annotated invariant. We believe that this teaching method can be deployed in massive open online courses (MOOCs). Finally, we evaluate our approach on several examples of relatively small assembly programs.

We make the following contributions in this paper:

- A sound program verifier for proving properties about 32-bit x86 programs
- An algorithm for generating feedback about incorrect student annotations, and candidate locations that need annotations
- Evaluation on several examples, each of which require multiple annotations from the student



```
invariant forall i.
((i >= 1 && i < esi)  =>
  *(0x00112000 + i * 8) == 0x00400387) &&
!(i >= 1 && i < esi)  =>
 *x == old(*x)
```

```
forall i. i >= 1 && i < 2048
=>
 *(0x00112000 + i * 8) ==
   0x00400387)
```

```
0x10056d mov  %ecx,0x112000(,%esi,8)
0x100574 mov  %ebx,0x112004(,%esi,8)
0x10057b add  $0x1,%esi
0x10057e cmp  $0x7ff,%esi
0x100584 jle  10056d <setup_ept+0xc7>
```

Figure 1: Student interaction

2 Student Interaction and Pedagogy

The student interacts with assembly code provided to them, while developing a set of assertions and loop invariants that allow verification of an instructor provided property. They interact through a web interface that presents panels with the assembly code and property provided by the instructor, annotations added by the student, and feedback to the student from the verifier. This interaction is illustrated in Figure 1 with an example of a property for a micro hypervisor.

Initially, the student will ask SILVA to verify the code as it is. It will likely not be able to do this, and the student will then be prompted to add annotations about the code. In describing their annotations, the student uses a C-like grammar where registers can be operated on and compared with other values. Additionally they are allowed quantifiers over 32-bit values.

Any time the student adds assertions or loop invariants, SILVA will check that they are true. If they do not hold, the student will be presented with a counterexample set of register and memory values and an execution trace that leads to that counter example. The student can step forwards or backwards through this execution trace highlighting the current instruction and seeing the applicable register and memory values. The student tries to correct the assertion or loop invariant and then asks for reverification.

If all student assertions or loop invariants are valid, SILVA tries to verify the program property. If it does not hold, the student will again be presented with counter example values and an execution trace. The student can add additional assertions or loop invariants to allow the verification.

If SILVA is able to verify the instructor property, the proof success is displayed to the student. The need to assist the theorem prover in verification forces the student to interact closely with the assembly code.

3 Related Work and Tools

There have been few related efforts, in both program verification and MOOCs. Brumley et al. [2] perform symbolic execution of user-level binary programs in order to find exploits. In this work, we build on their conversion from assembly program to a RTL-like intermediate representation. However, we do not perform symbolic execution because we would like students to provide annotations and prove properties about the

x86	BAP IR
`mov $0x3, %edx`	`R_EDX:u32 = 3:u32`
`mov $0x4, %eax`	`R_EAX:u32 = 4:u32`
	`T_t1:u32 = R_EAX:u32`
	`T_t2:u32 = R_EDX:u32`
`add %edx, %eax`	`R_EAX:u32 = R_EAX:u32 + T_t2:u32`
	`R_CF:bool = R_EAX:u32 < T_t1:u32`
	`R_OF:bool = high:bool((T_t1:u32 ^ ~T_t2:u32) & (T_t1:u32 ^ R_EAX:u32))`
	`R_AF:bool = 0x10:u32 == (0x10:u32 & (R_EAX:u32 ^ T_t1:u32 ^ T_t2:u32))`
	`R_PF:bool =`
	`~low:bool(R_EAX:u32 >> 7:u32 ^ R_EAX:u32 >> 6:u32 ^ R_EAX:u32 >> 5:u32 ^`
	`R_EAX:u32 >> 4:u32 ^ R_EAX:u32 >> 3:u32 ^ R_EAX:u32 >> 2:u32 ^`
	`R_EAX:u32 >> 1:u32 ^ R_EAX:u32)`
	`R_SF:bool = high:bool(R_EAX:u32)`
	`R_ZF:bool = 0:u32 == R_EAX:u32`
`mov %eax, 4(%esi)`	`mem:?u32 = mem:?u32 with [R_ESI:u32 + 4:u32, e_little]:u32 = R_EAX:u32`

Figure 2: x86 instructions and the BAP IR which is generated from them.

programs. Klein et al. [4] perform formal verification of a microkernel, by proving properties on high-level model, and orthogonally proving correctness about the compiler translation to executable binary.

In relation to other MOOC efforts, Fast et al. [3] deployed a MOOC on formal reasoning. The goal is to have students prove lemmas by starting from a set of assumptions and axioms, and performing rewrite rules to arrive at the goal. Their motivation is to teach students about inference rules in logic. In our case, we ask students for program invariants because our motivation is to teach students about x86 assembly.

BAP Though initially designed for bugfinding and not formal verification, BAP has wide coverage of the x86 semantics. We use BAP's `toil` front-end and its intermediate representation (the *BAP IR*) to create a formal model of the program. BAP has three general statement types that we use. `Move` statements transfer a value from some location into a variable. `Store` statements transfer a value into a memory array. `Jump` statements redirect program flow. Each statement may contain multiple *expressions*, which can either be `Load`s of an array at a specific index, or any of a number of unary or binary operations on other expressions.

Figure 2 gives a simple x86 program which adds two integers and stores the result at the address of register `esi` plus 4. The corresponding BAP IR is shown at right, with all flags updated as the x86 specification requires. Finally the result is stored into memory.

Boogie Used as a frontend for theorem provers, Boogie [1] is "an Intermediate Verification Language (IVL) for describing proof obligations to be discharged by a reasoning engine, typically an SMT solver." It provides a programming language interface that is easier to work with than raw SMT solver clauses. SILVA takes as input the BAP IR and produces a Boogie program. We then use Boogie and a SMT solver, Z3 [5], to check the students work.

4 Algorithm

This section describes how SILVA verifies the student assertions and returns either success or one of the three error conditions: assertion failure (on student annotations), property failure, missing loop invariants.

There are three inputs to our main algorithm.

Binary program, as BAP IR. The first step in the process is to generate the BAP IR from the x86 binary. We use The `toil` utility provided in BAP to achieve this goal. We assume that the input program is provided as a well-formed ELF. This allows BAP to extract the instructions from the binary at the correct offsets.

Student annotations. Given the binary program, the student writes annotations to go along with the program. These annotations can either be assertions which must hold true at different locations in the program, or invariants which hold for loops.

Instructor property. Finally, the instructor provides a property which the code satisfies, but which Boogie cannot prove without annotations. This property is written in the same language as the student annotations.

4.1 Algorithm Overview

There are four main steps to our algorithm, illustrated in Figure 3.

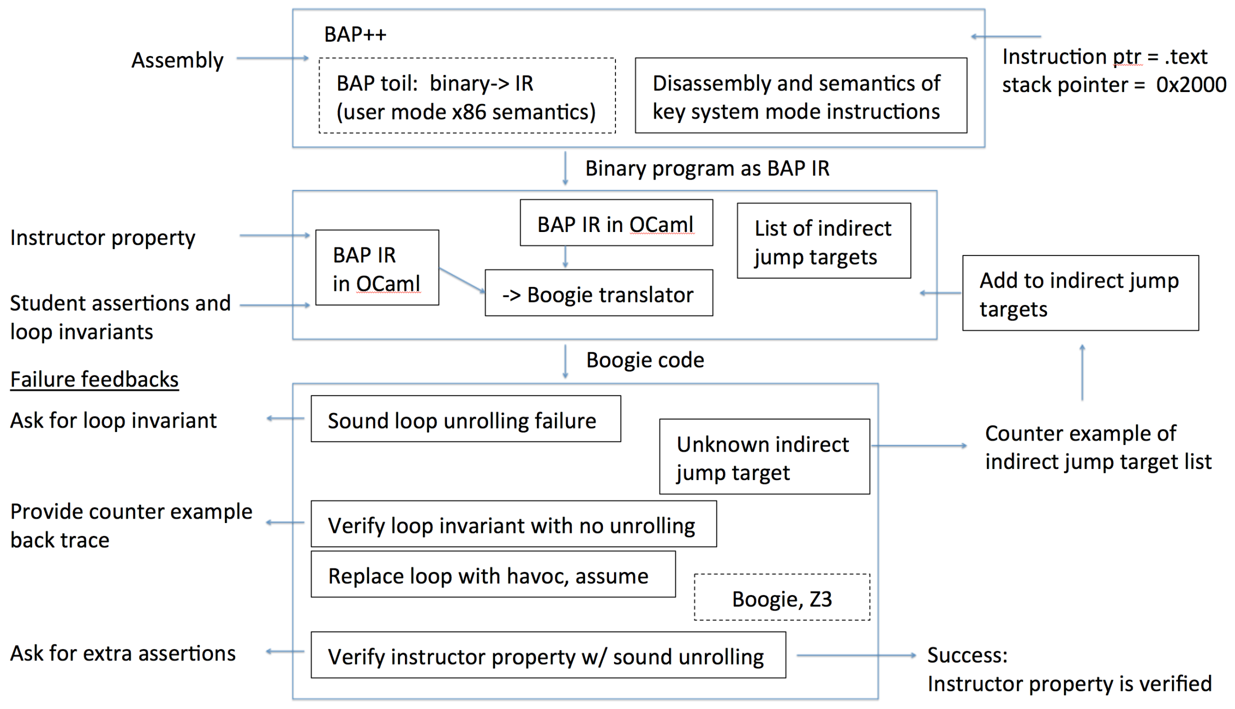


Figure 3: Algorithm stages

Construct a formal model of x86. We do not define a formal model of x86 directly, but of the BAP IR. We then rely on BAP to correctly transform x86 to the BAP IR. Our formal model includes a memory array, each of the the 32-bit registers, the unary and binary operations on registers, and functions for loading and storing different bit-widths out of memory. We also handle the compilation of conditional and indirect jumps, which we describe in detail in Section 4.3.3.

Identify a loop which requires an invariant. The first step in the process is to identify a loop that requires a loop invariant. To do this, we use *sound* loop unrolling [1] and attempt to prove the final property. If verification succeeds, we are finished. Otherwise, if we encounter an assertion failure on the sound loop unrolling, we have identified a candidate loop which is likely to have too many iterations to unroll and verify efficiently. We return with an error asking the student for an invariant at this loop.

There exists one further complication for programs which contain indirect jumps, see Section 4.3.3.

Verify that the invariant is true. After identifying a loop which requires an invariant, and receiving the invariant from the student, we attempt to verify that the invariant is in fact true. We disable loop unrolling to do this, since unrolling the loop will also unroll any invariant checks, and check the invariant for only a finite number of iterations instead of doing so inductively.

We do this by splitting the single invariant in to two assert statements and one assume statement. The first assert is inserted directly before the loop body, to verify that the loop invariant is true upon entry to the loop. Then, we add an assumption of the invariant to the beginning of the loop, along with an assertion that the invariant holds true at the end of the loop. These two statements verify that the invariant holds inductively.

If the invariant does not verify, then we return this error case to the student and ask them to correct the invariant so that it is true.

Rewrite binary to use that invariant. Once we have an invariant which is true, we can remove the loop and replace it with a havoc on all variables updated by the loop, followed by an assumption that the loop invariant is true. We can then return to the first step and identify another loop which requires an invariant, or can attempt to prove the property again.

Check that property is true. After we have all the invariants we need, we check that the property provided is in fact true. We do this by again enabling sound loop unrolling on the modified binary with all large loops removed and replaced with their invariants.

4.2 Large Loops

4.2.1 Finding Loops

The first step in removing large loops from a program is to identify the location of a loop. Because x86 binaries are not structured programs, this is a nontrivial problem.

We use a strict definition of a loop in order allow a simpler analysis procedure. We define a loop in the control flow graph as a pair of nodes (A, B) such that there is a conditional edge from B to A, all paths that

[1] Sound loop unrolling first unrolls the loop to a fixed depth, but then inserts an assertion that the loop is not executed more times than it is unrolled.

```
   movl $10, %eax
   movl $10, %ecx
   xor %ebx, %ebx
l: // assert Forall x . (x >= 10 && x < eax) => *x = 0
   // assert eax+ecx = 20
   // assume Forall x . (x >= 10 && x < eax) => *x = 0
   // assume eax+ecx = 20
   movb %ebx, (%eax)
   incl %eax
   decl %ecx
   // assert Forall x . (x >= 10 && x < eax) => *x = 0
   // assert eax+ecx = 20
   jne l
```

Figure 4: Simple example program with loop invariants expanded.

pass through A eventually pass through B (that is, B post-dominates A), and all paths which reach B have passed through A (that is, A dominates B).

4.2.2 Verifying Loop Invariants

Figure 4 shows an example program as well as the loop invariants to prove that it writes 0 to the bytes from 10 to 20. We use two loop invariants. First, we use one invariant to show that `eax` and `ecx` always sum to 20. Then, we use the other invariant to show that zero is always written to every memory value smaller than `eax` but greater than 10.

We split each invariant into two asserts and an assume, as described earlier. Upon entering the loop, the two registers both are initialized to 10 so the property is true. Also, `eax` is 10, so no value x is both greater than or equal to 10 and less than 10 simultaneously, so this invariant is trivially true.

Both properties are then assumed to be true. The program writes 0 to the memory pointed at by `eax`, increments `eax`, and decrements `ecx`. This leaves the first property true, as the sum is still 20. The second property is also true because we have incremented `eax`, but also written 0 to one new address.

So this property is true, and together these two properties can be used to show that for all values between 10 and 20, the memory at that address is zero.

4.2.3 Replacing Loops with Invariants

Because of our strict definition of a loop, after finding one, we can remove all nodes internal to the loop body. We do this by again defining A as the loop head and B as the loop tail, and removing all nodes reachable from A before reaching B, as well as A and B themselves. Our loop definition ensures that this set of nodes can neither jump to any other point, nor is jumped to by any other point.

We then havoc every value updated by this loop body, and insert our loop invariant (which was proven to be true) as an assumption.

```
function ITE<a>(b : bool, x : a, y : a) returns (a);
axiom (forall <a> b : bool, x : a, y : a :: {ITE(b,x,y)} (b ==> ITE(b,x,y) == x) && (!b ==> ITE(b,x,y) == y));
function align(addr : bv32) : bv32;
axiom (forall addr : bv32 :: {align(addr) : bv32} align(addr) == addr[32:2] ++ 0bv2);

function STORE_LIT_8(mem : [bv32] bv32, addr : bv32, value : bv8) : [bv32] bv32;
axiom (forall mem:[bv32]bv32,addr:bv32,value:bv8 :: {STORE_LIT_8(mem,addr,value) : [bv32]bv32}
  STORE_LIT_8(mem,addr,value) ==
  ITE(addr[2:0] == 0bv2,mem[align(addr) := mem[align(addr)][32:8]++value],
  ITE(addr[2:0] == 1bv2,mem[align(addr) := mem[align(addr)][32:16]++value++mem[align(addr)][8:0]],
  ITE(addr[2:0] == 2bv2,mem[align(addr) := mem[align(addr)][32:24]++value++mem[align(addr)][16:0]],
  ITE(addr[2:0] == 3bv2,mem[align(addr) := value++mem[align(addr)][24:0]],mem)))));
```

Figure 5: Boogie model for storing 8-bit integers to memory

4.3 Boogie Model

4.3.1 Converting the BAP IR to Boogie

The main component of SILVA converts the BAP IR to Boogie, which can then be converted to an SMT formula and sent to the SMT solver (Z3 in this case). BAP and boogie have different constructions and expressiveness, and converting from one to the other has several nontrivial steps.

The register-transfer language of BAP maps well onto Boogie's use of typed variables. Additionally, the BAP memory model maps directly to an array of 32-bit bitvectors in Boogie. We model all jumps within the binary as `goto`s within Boogie, and conditional jumps as nondeterministic `goto`s. We describe in section 4.3.3 how we deal with indirect jumps.

When converting a conditional jump to a nondeterministic goto, there is slightly more to do than just converting `cjmp(Condition, Lab1, Lab2)` to `goto Lab1, Lab2`. We must observe that `Condition` holds when taking the first path, but does not hold when taking the second path. It would be incorrect to make the code at `Lab1` begin by issuing an `assume` of `Condition`, because there may be many entry points to `Lab1`. We therefore convert the conditional jump to a sequence of Boogie instructions which nondeterministically jumps to one of two temporary locations, each of which first assumes either the condition or its negation, and then jumps to the correct address. This allows the correct assumptions to be made along each code path.

Boogie also has support for constructs such as loops (with invariants) and procedures (with pre- and post-conditions). We do not use these because the input BAP IR is an unstructured program.

4.3.2 Formal Boogie Model of BAP IR

We construct a formal model of the BAP IR in Boogie. We do not need to model any x86 semantics because at this point every effect of the x86 instructions has been modeled in the simpler BAP IR.

Registers are modeled using 32-bit bitvectors in Boogie, when the full width of the register is used (e.g., `eax` is read), or 16-bit or 8-bit bitvectors if requested (e.g., `ax` or `ah` is read).

We then create a model of each of the unary and binary operations present in the BAP IR (e.g., addition, subtraction, logical and), which operate on any of 8-, 16-, and 32-bit operands. When possible, we have Boogie use the underlying SMT solver's operations instead of reimplementing basic operations on bitvectors within Boogie.

Our model of memory can be thought of as a map from 32-bit bitvectors to 32-bit bitvectors. This allows Boogie to only model those memory addresses which have been written to, and not reason about all 2^{32} possible addresses.

The reason we choose this model instead of a map from 32-bit bitvectors to 8-bit bitvectors is performance: the majority of reads and writes are to 32-bit integers on 4-byte aligned words. This model allows us to simply load and store entire bitvectors at once.

However, this complicates the process for loading and storing values which are not 32-bits. In particular, when storing a 16-bit value we must first load the full 32-bit value from memory, overwrite the correct 16-bit part, and store the value back to memory.

Also note that while memory is technically mapping from 32-bit bitvectors to 32-bit bitvectors, in reality it is mapping from 30-bit bitvectors to 32-bit bitvectors, since the low two bits on the source are always zero.

Figure 5 has the Boogie model we constructed for storing 8-bit bitvectors into memory. It begins with the definitions of ITE (if/then/else) as picking either the second or third arguments based on the condition. Then the align function is defined to align a 32-bit value to 4-byte word boundaries. Then the actual store function is defined case by case for each of the 4 bytes that could be written to within a word. Because x86 is little-endian, an aligned write overwrites the high byte of the 32-bit word.

4.3.3 Handling indirect jumps

Indirect jumps present a challenge when attempting to verify the correctness of an assertion. Because we are verifying the program as a SMT formula, we must have a statically known control flow graph.

To handle this issue, we convert each indirect jump to an equivalent form which does not use an indirect jump. Naively, one might consider allowing every possible address as a valid jump target. However, this does not work for two reasons. Boogie requires that the control flow graph of all programs be *reducible*, and it is very likely that by allowing an indirect jump to target an instruction this program will not be reducible. However, more generally, allowing this possibility would significantly increase the size of the SMT formula.

Instead, SILVA uses a counterexample-guided approach to identify all possible jump targets for each indirect branch. We create a set of possible jump targets for each indirect jump. This set is initially assumed to be empty (i.e., that this indirect jump can go to no other instruction). When compiling the BAP IR to a Boogie program, we use this set as the set as the possible jumps in our switch statement. If the actual jump target is not a member of the set, we throw an assertion error. We then generate the Boogie program.

We then execute the Boogie program. If an assertion failure occurs, and it was due to one of our indirect jump assertions, we have found a new possible jump target. We then add this target to the set of possible targets for that indirect branch, regenerate the Boogie program, and try again. If we do not encounter an assertion failure because of the indirect jump assertions, it means we have found an over-approximation of the possible jump targets.

Our procedure is sound, modulo instruction aliasing. That is, we assume that indirect jump never targets the middle of an existing instruction.

4.4 Generating Counterexamples

Our tool provides students with counterexamples whenever an error is returned. We do this by adding additional instrumentation to the Boogie representation of the BAP IR. We create a map of 32-bit integers for each of the registers we wish to track. Immediately before each instruction, we insert each register into its corresponding map at the next available index.

Whenever an assertion fails or the final property does not verify, Boogie returns a counterexample with the assignments of all variables, including these register maps. We can use these maps to create a trace of the state of the CPU at each instruction address, and provide this to the student in the form of a counterexample trace.

4.5 Limitations

There are two cases in which SILVA will fail in a program. First, we make the assumption that all large loops satisfy our dominance property. If this is not the case, we will not identify the loop, and will not be able to remove all nodes internal to the loop, without removing instructions used elsewhere. This implies that large loops can not contain function calls, because the return instruction is an indirect jump. Second, we assume that loops do not contain indirect jumps.

We do not introduce unsoundness if either of these two properties are not satisfied. We will simply report that one of these two properties does not hold, and abort.

5 Discussion

We give an informal argument for why our approach is sound. We divide this argument into two parts: 1) our method for checking each student annotation is sound, and 2) our method for checking the instructor property is sound.

At any phase in our verification, we always use an over-approximation of the control flow graph. Each known jump target is encoded as a valid target location of a goto statement. For unknown jump targets, we replace them with the $assert\ false$ at the source locations. Therefore, if at any time, the solver discovers a new jump target, an assertion failure indicates the new jump target. We add this to our set of known jump targets. Since we always maintain an over-approximation of the control flow, we never unsoundly prune control flow behaviors.

Once we prove that a student annotated loop invariant is correct, we replace the loop body with that invariant ($havoc$ followed by $assume$). For soundness, we must show that our method for checking each student annotation is sound. Since we use sound loop unrolling (in addition to the inductive check of the loop invariant), we guarantee that we always explore an over-approximation of all program behaviors. The same argument applies for checking the instructor property.

6 Evaluation

We evaluate our approach on several assembly programs, some of which are created by us. The assembly programs have up to 15 lines of C code, which translate to approximately 300 lines of Boogie code. In each experiment, we have at most 2 loops in the program, each of which need loop invariants. In some cases, we need multiple auxiliary invariants for the same loop.

Checking each annotation takes about 2 seconds running on a commodity machine. In total, checking all annotations and the final program property takes up to 5 seconds on all our benchmarks.

7 Conclusion and Future Work

This paper introduces SILVA , a tool allowing students to work together with a formal analysis framework in order to prove properties about binary programs, to further the student's understanding of the code. We demonstrate that proving properties about binary programs is possible with student assistance. This is achieved through use of the BAP framework to convert x86 to a simplified IR, which is then compiled down to Boogie and finally a SMT formula.

We identify multiple difficulties in compiling programs to Boogie programs, including handling loop, conditional jumps, and indirect jumps. We resolve all of these issues automatically, with the exception of large loops which require invariants.

We believe this tool can be used to effectively teach students to understand x86 programs more completely, as well as learn about the advantages of formal verification.

References

[1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag.

[2] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] E. Fast, C. Lee, A. Aiken, M. S. Bernstein, D. Koller, and E. Smith. Crowd-scale interactive formal reasoning and analytics. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 363–372, New York, NY, USA, 2013. ACM.

[4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[5] L. Moura and N. Bjrner. Z3: An efficient SMT solver. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

Online Monitoring of Signal Temporal Logic for Virtual Lab Autograding

Shromona Ghosh
in collaboration with Garvit Juniwal, Alexandre Donzé, Sanjit Seshia

May 16, 2014

Abstract

Massive Open Online Courses(MOOCs) are online courses which are designed to scale to a large number of participants who do not need to be registered at an academic institution. Today, the participation is in tens of thousands. Due to this huge volume of participants, we face a lot of problems. The course we have been concentrating on is Berkeleyx 149 *Introduction to Embedded Systems: A Cyber-Physical Systems Approach* provided by the University of California, Berkeley on EdX. This course comes with labratory exercises where students are expected to work with the CalClimber in a online virtual lab environment. For grading, we simulate the robot in the LabVIEW Robotics Environment Simulator. The instructor can define certain goals which the students need to satisfy in the form of Signal Temopral Logic formulae and then use BREACH, to check if the students solution satisfies this property. We currently use an offline monitor, which requires us to already simulate the robot before grading. By incorporating an online monitor to the simulator, we can reduce simulation costs and get a much quicker results.

1 Introduction and Motivation

1.1 Need for autograding in MOOC

In the classroom version of the EECS149 'Introduction to Embedded Systems', course materials are covered in the video lectures along with the books assigned for reading. These lecturse are available online and the textbook is available as a free PDF online. The laboratory exercises are aimed to give students experience with three distinct level of embedded software design, programming within a real-time operating system and model based design for cyber-physical systems. The laboratory exercises follow the textbook *Introduction to Embedded Systems: A Cyber-Physical Systems Approach* exercises. Extending the laboratory exercises to scalable online versions is an important cause of concern.

Due to the huge participation seen in MOOCs, we face a lot of issues regarding it's scalability. We concentrate on the BerkeleyX 149 'Introduction to Embedded Systems' course taught by Lee & Seshia, University of California Berkeley. For laboratory exercises, students are required to write codes which are to be run in robots in different lab environments. While in the classroom version of this class, students were able to download the program into a real robot. Grading was done based on the obstacles the student could overcome in a given environment. Extending this to student codes submitted in the online version of the code, is not a feasible option. In [3], they propose that for a cyber-physical lab, only simulation seems a viable option. In the BerkeleyX 149 course, students are expected to download their code into the Cal Climber in

LabVIEW Robotics Environment Simulator. The Cal Climber exercises require a gcc compiler for development of the C controller and LabVIEW for the simulation environment and development of the dataow controller.

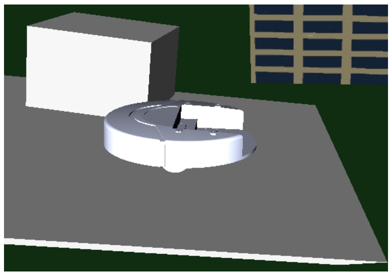


Figure 1: The figure above, shows the CalClimber in a virtual lab as part of the online course on Embedded Systems provided by UC-Berkeley

Providing an online simulator solves only part of the issue. We need to find a quick way to grade the student solutions and provide feedback to the students as well. We use an autograder for student garding. Currently, we simulate the student solution on a robot in the virtual lab and then pass the generated trace on BREACH. We define goals in terms of Signal Temporal Logic formulae. BREACH uses algorithms for offline monitoring of STL on the traces to calculate robustness value of the STL property on the trace, and can thus assign grades to the students. Based on when and degree of satisfaction and violation of a STL property on the given trace, we can give feedback to the student.

1.2 Drawbacks of Offline Monitoring of Signal Temporal Logic

Offline monitoring is implemented on an execution trace obtained post simulation. Simulation process involves solving ODE's, this can be an expensive process. Running a simulation for say 60 seconds can be very inefficient and expensive, especially in cases where the STL formula maybe satisfied or violated in the initial bits of the trace.

Also, in BREACH [2], we make an assumption that the trace remains a constant after the end of the trace i.e every postfix will always be a constant with a value equal to the last data point.

1.3 Advantages of Online Monitoring

Online Monitoring is a demand driven approach. It asks for more traces only if no decision can be made with the amount of trace present. By implementing it as part of simulator, we can exit from the simulator once the property is satisfied or dissatisfied. This would help reduce simulation time and cost.

By replacing a single robustness value with a robustness interval, we don't need to make any assumption on the postfix of a trace as we did in BREACH.

1.4 Related work

Offline monitoring of STL has been implemented in [2] as a part of BREACH. This paper also introduces linear time algorithm for calculating robust values for all STL operators. However, this work has all the drawbacks of offline monitoring as stated above. No prior work has been done as part of online monitoring of STL.

Online monitoring of linear temporal logic(LTL) has been discussed in [1] and [5]. Online monitoring of metric temporal logic (MTL) has been discussed in [4]. The paper talks about maintaining a 2D array containing satisfaction values for all nodes in a STL formula and updating them with any incoming data. This paper also discusses how we need to maintain only a window of values for a a given operator based on its interval.

2 Problem Definition

We propose to investigate online monitoring of STL formulas and see how it would improve the working of the autograder. We plan to implement and execute a online STL monitor together with the simulator of. Such a monitor can, e.g., trigger actions in case the formulas is satisfied or violated, as soon as it is the case. In our case, if a formula is violated, it gives feedback to the student on where they went wrong and how they can fix it. Grading is given based on how much the property is satisfied and when it is satisfied.

We propose to formalize the semantics of online STL monitor. We define robust intervals and how they are an improvement over the traditional robustness values.

3 Approach

3.1 Boolean and Quantitative semantics of Signal temporal logic

The boolean and quantiative semantics of signal temporal logic has been discussed in [3].

3.1.1 Boolean Semantics of Signal Temporal Logic

For a trace 'w', the validity of an STL formula φ at a given time 't' is given as the following:

$$w, t \vDash true$$

$$w, t \vDash x_i \geq 0 \qquad iff x_i^w(t) \geq 0$$

$$w, t \vDash \neg\varphi \qquad iff w, t \nvDash \varphi$$

$$w, t \vDash \varphi \wedge \psi \qquad iff w, t \vDash \varphi \, and \, w, t \vDash \psi$$

$$w, t \vDash \varphi U_I \psi \qquad iff \, exists \, t' \in t + I \, s.t \, w, t' \vDash \psi \, and \, for \, all \, t" \in [t, t'], w, t" \vDash \varphi$$

Other operators can be defined based on these basic operators.

$$false := \neg true$$

$$\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$$

$$\Diamond_I \varphi := true U_I \varphi$$

$$\Box_I \varphi := \neg \Diamond_I \neg \varphi$$

The satisfaction signal is given χ(φ,w,t) at a time 't' is given as:

$$for\, all\, t \in dom(\varphi, w), \qquad \chi(\varphi, w, t) := \begin{cases} \top, & \text{if } w, t \vDash \varphi; \\ \bot, & \text{otherwise.} \end{cases} \tag{1}$$

We break a given STL formula into a tree like structure where the terminal nodes are atomic predicates and the each parent node is an operator. We calculate the satisfaction signal of each node starting from the terminal nodes and then move up to the root node which is the complete STL formula. Even though this tells us if a STL formula is satisfied, it is not enough. We need some quantitative information which we can use for decision making.

3.1.2 Quantitative Semantics for Signal Temporal Logic

For a trace 'w' and formula φ, the quantitative semantics ρ(φ,w,t) at a time 't' is given as:

$$\rho(true, w, t) = \top$$

$$\rho(x_i \geq 0, w, t) = x_i^w(t)$$

$$\rho(\neg\varphi, w, t) = -\rho(\varphi, w, t)$$

$$\rho(\varphi \wedge \psi, w, t) = min \rho(\varphi, w, t), \rho(\psi, w, t)$$

$$\rho(\varphi U_I \psi, w, t) = \underset{t' \in t+I}{sup}\, min\{\rho(\psi, w, t'), \underset{t" \in [t,t']}{inf}\, \rho(\varphi, w, t")\}$$

$$\rho(\Diamond_I \varphi, w, t) = \underset{t' \in t+I}{sup}\, \rho(\varphi, w, t')$$

$$\rho(\Box_I \varphi, w, t) = \underset{t' \in t+I}{inf}\, \rho(\varphi, w, t')$$

Here, the atomic predicates do not evaluate to a $\top$ or $\bot$, but give a real value representing the distance to satisfaction or to violation. This gets propagated to the other operators in the formula based on the above quantitative semantics. The satisfaction signal can now be defined as

$$\chi(x_i \geq 0, w, t) = \begin{cases} \top, & \text{if } x_i^w(t) \geq 0; \\ \bot, & \text{otherwise.} \end{cases} \tag{2}$$

ρ is known as the robustness value and is a scalar quantity. The robustness estimate can be to used generate feedback. This has been implemented in Breach as offline monitoring of STL properties. We see that we can implement quantiative semantics only, as we get all the information about a formula's satisfaction and violation from it's robustness value.

3.2 Formalizing the new approach

We break down online monitoring of STL properties into 3 parts.

3.2.1 Robust Intervals

We replace the robustness values with an robust interval. This contains more information than a single robust value in the case of offline monitor. It gives us a range within which the final robustness value will lie in. We observed that there are 5 different ranges:
1. [-∞, +∞] : This implies that no information is known yet about the STL property. The robustness value can be anything. This is the initial case, when no data is known.
2. [-a, +b] : This is a non-trivial interval. This interval implies, even though some amount of data is known, it's not sufficient to say if the STL property is satisfied or not. We could demand for more data. However, even if we don't have anymore data, we have a smaller range of the robustness value. On knowing anymore data, this interval becomes smaller.
3. [a, +∞] : This is the positive half interval. If an STL property reaches this interval, this means the property is satisfied. Knowing anymore data will give us the final robustness value, but we can stop simulation here.
4. [-∞, -a] : This is the negative half interval. If an STL property reaches thi interval, this means the property has been violated. If we know anymore data, this would reach its final robustness value.
5. [a,a] : This is the triavial value. 'a' is the final robustness value. If it is positive, the property is satisfied. If it is negative, the property is violated.

We have [a,b] and [-a,-b] as well. They are a smaller range of 3 and 4 respectively. They have the same significance as the corresponding half interval.

The functions governing the limits of the intervals are monotonic. This implies, the intervals converge to give us a single robustness value. The intervals become smaller on finding out more data.

This is an improvement over having only a single robustness value. Imagine a case, offline returns a single robustness value. We do not know if this is the final robustness value, or it's a lower or upper limit. We can only know more about the robustness value, if we had more data. This is why BREACH assumes that the value is a constant. But a realistic approach to this would be to have an interval.

3.2.2 Memory Allocation

We treat bounded and unbounded operators differently. We can gaurantee that bounded operators always need finite memory. On nesting of operators, the innermost STL atom needs memory equal to the sum total of all the outer operators.

Let us look at the operators more closely,
1. For $G_{[a,b]}\varphi$ or $F_{[a,b]}\varphi$ from $[t_1, t_2]$ we need φ from $[t_1 + a, t_2 + b]$. We can gaurantee that the values of G or F reach there final robust value in this interval.
2. For $\varphi \wedge \psi$ or $\varphi \vee \psi$ or $\neg\varphi$ from $[t_1, t_2]$ we need φ and ψ from $[t_1, t_2]$.
3. For $\varphi U_{[a,b]}\psi$ from $[t_1, t_2]$ we need data for φ and ψ from $[t_1, t_2 + b]$. This is because we rewrite timed until as follows,

$$\varphi U_{[a,b]}\psi = (G_{[0,a]}\varphi U \psi) \wedge (F_{[a,b]}\psi)$$

Timed until also reaches the fixed robust value by the end of the interval.

In the parser, we topologically move through the formula tree starting from the root node and allocate memory to every node. At every node, we add memory based on the above rules and move forward to the leaf nodes.

3.2.3 Break Down of a STL formula

A given STL formula can be re-written as a tree, where every node correponds to a STL operator which can either be a boolean operators such as conjunction, disjunction and negation; and temporal operators such as bounded globally, bounded eventually and bounded until. The leaf nodes are STL atoms.

Consider the given STL formula,

$$\varphi_1 \vee ((G_{[a,b]}\varphi_2)U_{[c,d]}(F_{[e,f]}\varphi_3))$$

In STL semantics, we say a property is said to be satisfied, if it is satisfied at time 0. While breaking down the STL formula, we also allocate the amount of memory that needs to be reserved for every level. The above formula is broken down as shown below :
1. The top level operator i.e the root of the tree is $\vee$. We need it's value only at 0, so we allocate [0,0] to $\vee$.
2. $\vee$'s children are φ_1 and the timed until operator U. We thus allocate [0,0] to φ_1 and [0,0] to the until.
3. U's children are G and F. So we allocate [0,d] to both G as well as F.
4. G's child is φ_2, so we allocate [a,b+d] to φ_2.
5. F's child is φ_3, so we allocate [e,f+d] to φ_3.

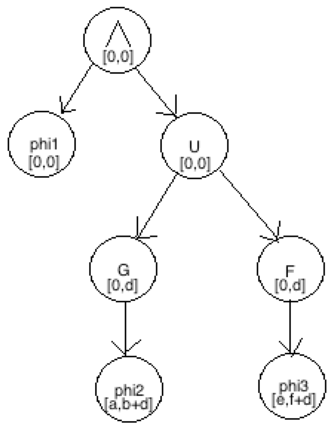


Figure 2: Breaking a STL formula and allocating memory

We allow a node to insert values into it's child as well as it's parents. The values inserted into the parent are considered to be 'important values'.
1. In the case of eventually and globaly, for an incoming data point, we insert the translated window begin and window end points. Suppose we have, $G_{[a,b]}\varphi$, if the incoming value of φ is at time t, then we insert, t-a and t-b into the window of the parent.
2. In the case of conjunction, disjunction and negation, no new values are inserted besides those appearing in the child.

3. In the case of until, we insert those time points where the 2 child nodes intersect along with the time point appearing in the child. This is done so that we can calculate untimed until in a linear fashion.

Values which are inserted in the child are merely interpolated values and are of no importance. They are inserted only to ease optimizations.
1. In the case of eventually and globally, if we are calcualting the value at t, we insert t+a and t+b into the child.
2. In the case of conjunction, disjunction and negation, we insert only those time points into the child which appear in one of the childs and not the other or those values which have been inserted into the window from it's parent.
3. In the case of until, we insert those values into the child which have inserted into this window because of it's parent.

The algorithms used for eventually, globally and until are similar to those in BREACH. However, we use the monotonicity of the intervals to avoid re-calculating values which have already become a trivial interval. These insertions are done so as to ease the optimization in online monitoring. By allowing the parent to insert these interpolated values into the child, we can use a counter to mark the last closed (trivial) interval.

When a new data point comes in, we update the tree starting from the child nodes to the root node. If we can make a decision on the satisfaction or violation of the root node, we can choose to stop simulation.

3.3 Treatment of unbounded operators

Using the above appraoch for unbounded operators, leads to assigning infinite memory. For unbounded operators, we can do one of the following :
1. We can have a maximum upper limit for unbounded operators. This is usually equal to the amount of time for which we simulate the robot.
2. We can use Buchi automaton to describe unbounded operators. However, we seldom have only unbounded operators in an STL formula and they are generally nested with bounded operators.
3. We have been able to establish algorithms for unbounded eventually and globally which needs to store only the maximum and minimum of the incoming values.

4 Results

As preliminary testing, we randomly generated 75 formulae consisting of 2 operators and 3 operators and tested them on a given 100 seconds trace.The random formula generator, produced formula where every operator had an interval which was a subset of [0,10].So in the case of 2 operator formulae we have maximum window size of 20 while for 3 operator formulae we have a maximum window size of 30. We recorded 2 important time points, one where we can make a decision about the STL formula and the second where we get a concrete value.

The following table shows average decision time and robust time i.e time at which we can make a decision and time at which we can get a concrete robust value for the 2 and 3 level operators. The first column shows the number of operators in the formula. The second column shows the average time it takes to make a decision, whiel the third column shows the average time needed to reach the final robustness value.

No of Opers	Avg Decision time	Avg Robust time
2	4.782666667	8.197466667
3	5.272933333	8.191466667

We notice that in most cases we can make a decision about a formula, much before we actually get the concrete robustness value. We see the decision time is about 60% of the time required for us to get the final robust values. This generally happened when a top level eventually operator was satisfied or a top level globally operator was violated or if the top level operator was a boolean operator . Also we notice that the while offline would have run the monitor for 100 seconds, we stopped much before.

We noticed that there were cases where the formula reached it's concrete value much before the end of the horizon of the formula. This is an important observation because this tells us there are times when we don't need to have the complete window information to make a decision. We can stop the monitor even before the end of the window.

In the following table we show 2 important observations, the number of cases where we could make a decision before getting the concrete robustness values and the number of cases where we reached the concrete robustness value before the end of the horizon. The first column gives us the number of the operators in the formula. The second column shows the number of experiments run. The third column shows the number of cases we were able to make a decision and could stop the simulator before getting the robust values. The fourth column shows the numebr of cases we were able to get the final robustness value before the end of the horizon.

No of Opers	No of formulae	Decision	Robustness before horizon
2	75	60	24
3	75	47	34

There were a few cases where we couldn't make a decision before getting the final robustness value. This was seen in cases where we detected a violation in a top level eventually operator or a satisfaction in a top level globally operator. We noticed that there were a considerably large number of cases where we didn't even have to run the monitor for the length of the formula.

We are now using student solutions obtained from the BerkeleyX 149 course as test traces and running the properties defined by the professor on it using the online monitor. We will then compare these results to those obtained with the current autograder.

5 Conclusions and Future Work

5.1 Conclusion

In this report we have formalized the following new concepts :
1. Robust intervals
2. Online monitoring of Signal temporal logic

Online monitoring of Signal Temporal Logic is an improvement over the offline monitor, as it makes no assumptions on trace data and gives us more information on the robustness value of an operator at a given time. We have tested the monitor on randomly generated formulae, and have gotten some promising results.

5.2 Future Work

Our monitor currently supports bounded operators. We would like to extend it to unbounded operators as well. We have thought up algorithms for bounded eventually and globally, and are yet to come up with an algorithm for until which is linear time and requires finite memory.

We will continue testing the monitor on all the solutions obtained from the students in the BerkeleyX 149 course. We aim to incorporate it with that already existing autograder we have for the BerkeleyX 149 course and use it for grading in future offerings of the course.

We currently consider inputs where the traces are 'noiseless' i.e they have fixed robustness value, we want to extend our monitor for noisy inputs as well. In such a case we would get a 'noisy' final robust value. In such a case, the final robustness value wouldn't be a single value. It would be an interval. We would have to study how the noise in the input affects the final robustness value.

References

[1] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. Technical report, 2007.

[2] Alexandre Donzé, Thomas Ferrère, and Oded Maler. Efficient robust monitoring for STL. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 264–279. 2013.

[3] Jeff C Jensen, Edward A. Lee, and Sanjit A. Seshia. Virtualizing cyber-physical systems: Bringing cps to online education.

[4] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 152–166. 2004.

[5] Thierry Massart and Cedric Meuter. Efficient online monitoring of LTL properties for asynchronous distributed systems, 2006.

Auto-Grading Dynamic Programming Language Assignments

Liang Gong
University of California, Berkeley
gongliang13@eecs.berkeley.edu

May 15, 2014

Abstract

One challenge for massive online education is to develop automatic grading system that provide useful feedback. We present a new method for auto-grading programming assignments of introductory JavaScript Courses. Our system takes as input the student's solution and instructor's reference implementation for concrete testing and profiling. Based on the collected information, the system automatically finds counter examples revealing bugs in given solution and provide feedback including grading, bad code practise as well as recommendation on bug locations.

The system uses symbolic execution technique to achieve high test coverage and statistical bug localization technique to pinpoint bugs. Program analysis technique is heavily used in our system to detect cheating, prevent infinite loops and detect inefficient code patterns. We have evaluated our system on real-world course assignments obtained from the Introduction to Data Structure course at UC Berkeley (CS61B). Our results show that our system can locate on average 60% of bugs in the top 5 lines of code recommended. With the help of our system, we also pinpoint bugs in publicly available code repository.

1 Introduction

For web programming, increasing computations are migrated to font-end web page to provide real-time service and smooth user experience. Many important applications (*e.g.,* Facebook, Gmail, Google Map *etc.*) are written in JavaScript. While other traditional programming languages such as Java and even C/C++ starts to compile their code into JavaScript so that their applications can run in a browser. So JavaScript is the *de facto* web assembly language. The rise of server-side JavaScript on Node.js also takes on more responsibility for back-end computation.

Massive open online courses (*abbr.* MOOC) attracts a lot of interests and attentions in providing high quality education to people worldwide. MOOC provides increasing opportunities for students and instructors across the world and thus attracts more and more people registering online courses. As a result, providing feedback for the enormous amount of homework submissions emerges as a challenge in the online classroom. In courses where the student-teacher ratio can be very high, it is impossible for instructors to personally give feedback to students which leads to a new research problem:

With the increasing number of student's programming assignments submission, can we automatically grade the student's solution while providing useful feedback?

In this paper, we present an automated technique to provide feedback for JavaScript introductory programming assignments. Our approach is a test-based technique that adopts symbolic execution technique to find counter examples and achieves high test coverage. Based on the runtime information collected from high coverage testing, the approach analyses those information to automatically recommend possible program bug locations to provide students with feedback about how the program goes wrong and where it should be changed to fix the bug.

The problem of providing feedback appears to be related to the problem of bug localization, but they are different mainly due to the presence of the *reference implementation* which is the correct code provided by the instructor. In conventional bug localization research, only 1) buggy program, 2) a set of test cases and 3) corresponding test output (*test oracles*) are given. Due to the absence of correct implementation, collecting test oracles are considered expensive as they are manually labelled by developers[1]. This limits the massive application of trace-based bug localization. While for auto-grading problem on the other hand, it is safe to assume that the reference implementation is always is available from the instructor and thus the bug localization technique can collect test oracles by runing the reference implementation with low extra cost.

We show the effectiveness of our approach by running our system on course assignments from real-world course as well as publicly available implementation of classical algorithm and data structures (*e.g.,* JavaScript implementation of quick sort on Github[2]). Then we use classical bug localization evaluation metric to measure the accuracy of our bug localization system.

The main contributions of this paper are as follows:

- Different from existing test-based auto-grading systems, we proposed a auto-grading approach based on concrete testing that can achieve high test coverage rate and recommend possible bug locations. Our system also solves some of the practical issues of concrete testing based auto-grading system (*e.g.,* cheating, and non-terminate programs).

- Our approach accepts reference implementation which may be a system call (without function source code). Existing formal method based auto-grading systems requires not only source code of reference implementation but also additional specification (*e.g.,* error model[5]) to assist reasoning on bug locations. Our approach is based on statistical analysis on runtime information that does not requires those detailed specifications and collecting which might be an extra burden on the users.

- Our implementation supports almost all of the advanced program constructs (*e.g.,* higher order functions etc.) in JavaScript. existing auto-grading system for high level languages(*e.g.,*[5] for Python) often only supports limited programming constructs.

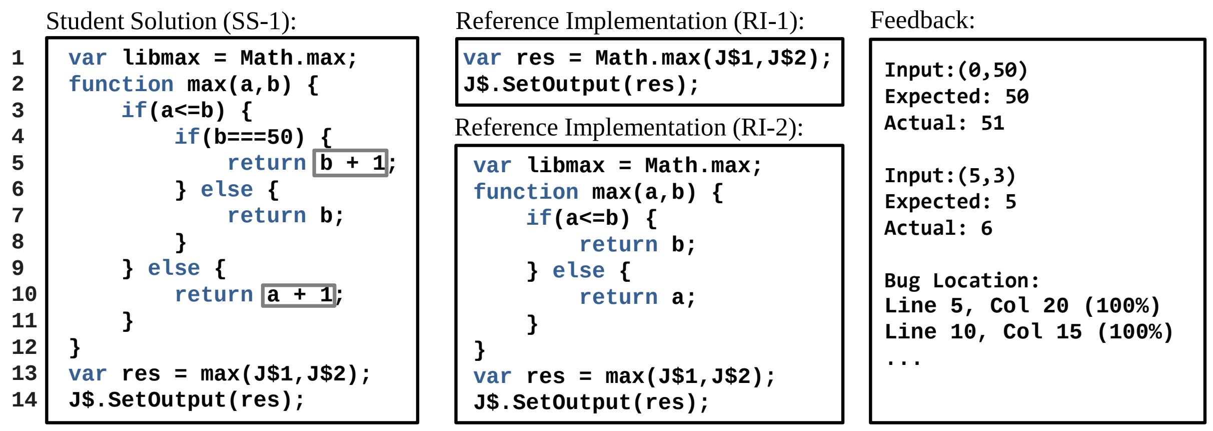
- We have evaluated our approach on assignments from real-world courses and publicly available code snippets. The evaluation demonstrates that our system can effectively find counter examples and generate feedbacks to assist auto-grading and online education.

2 Proposed Approach

Figure 2 shows the key building blocks of the proposed auto-grading system. The arrow flow indicates the data flow and the order in which each components are activated. In the beginning, when the *Student Solution* and *Reference Implementation* are provided by the user, our proposed system will use *test case generation* techniques including *symbolic execution*(section 5) and *random testing*(section 4) to generate

[1]This is known as the test oracle issue.

[2]`https://github.com/`



Student Solution (SS-1):

```
1   var libmax = Math.max;
2   function max(a,b) {
3       if(a<=b) {
4           if(b===50) {
5               return b + 1;
6           } else {
7               return b;
8           }
9       } else {
10          return a + 1;
11      }
12  }
13  var res = max(J$1,J$2);
14  J$.SetOutput(res);
```

Reference Implementation (RI-1):

```
var res = Math.max(J$1,J$2);
J$.SetOutput(res);
```

Reference Implementation (RI-2):

```
var libmax = Math.max;
function max(a,b) {
    if(a<=b) {
        return b;
    } else {
        return a;
    }
}
var res = max(J$1,J$2);
J$.SetOutput(res);
```

Feedback:

```
Input:(0,50)
Expected: 50
Actual: 51

Input:(5,3)
Expected: 5
Actual: 6

Bug Location:
Line 5, Col 20 (100%)
Line 10, Col 15 (100%)
...
```

Figure 1: A Running Example.

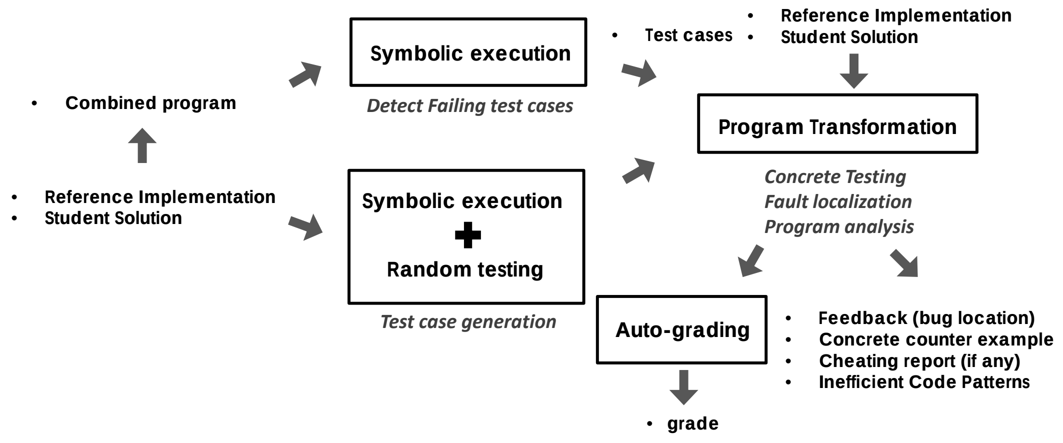


Figure 2: Approach Overview.

a specified number of test cases for concrete testing. To enhance the probability that our system detect a abnormal behaviour when there is a bug in the student solution, our proposed system adopts a technique to generate a counter example to trigger bugs (see section 8). Once all test cases are collected, we use program instrumentation (section 3) to profile the program execution under those generate test cases, the system also execute the exact same test input on the reference implementation to collect test oracles. Once test oracles and runtime information (*i.e.,* program spectra) are collected, we perform statistical fault localization (section 7) and get possible bug locations in the student's solution. During the execution of the instrumented program, our system monitors the execution to detect cheating, infinite loops and inefficient code patterns (section 9). Finally the bug location, counter examples, cheating information, inefficient code patterns (if any) and final grading are gathered and report to the user. In the following sections, we briefly introduce the rational and working mechanism behind those key components.

3 Dynamic Analysis Framework

Dynamic program analysis is the analysis of software applications by running the programs with a set of test inputs and analysing their behaviours during executions. This technique is widely used for software testing, software profiling, understanding software behaviours and many other applications of interest.

Conventionally, implementing a dynamic analysis framework for a dynamic programming language modifies the underlying virtual machine [35] to intercept and monitor low-level events. This approach however, has the following disadvantages:

Not all virtual machines are open source Some JavaScript virtual machines are not open source, or have strong license, which prohibits access to the source code or modifying the virtual machine for any purpose freely.

Hard to maintain the analysis framework code JavaScript virtual machine vendors has no obligation of preserving the inner function/method interface of the virtual machine. Consequently the previous developing effort are in vain if the dynamic analysis framework depends on inner functions which are removed or modified in the future.

Different programming paradigms Modifying the JavaScript virtual machines makes it difficult for a third-party developer to build the analysis module. JavaScript developers might not be familiar with the C/C++ programming paradigm and does not necessary have the background knowledge and experience of handling low level mechanism in which a glitch may crash the system.

In contrast, source code transformation makes the hooks and analysis code completely decoupled with the low level virtual machine and addresses all of the above difficulties. Further more, manipulating the source code makes the framework easily portable to any virtual machine that implements ECMAScript standard [13]. (*e.g.,* Mozilla SpiderMonkey [41], TraceMonkey [42] and Rhino [34], Nitro [15], Google V8 [14] used by Chrome and Node.js [16]*etc.*).

Moreover, our goal is doing dynamic program analysis on the fly in the browser. So the proposed approach instruments the program rather than modify VMs.

To illustrate the concept of code instrumentation, consider the following code snippet:

```
var a = b + c;
```

After instrumentation, the program is transformed into:

```
var a = W(B(R(b, 'b'), R(c, 'c'), 'b', 'c'), 'a');
```

Function `R(b, 'b')` means the callback function (*i.e.,* hook) that monitors the reading operation of variable `b`, the parameters of the callback function include the variable name and value. Similarly callback function `W` and `B` are for variable write and binary operations respectively. Inside each of those functions we will implement the semantics of the original JavaScript code and call an additional function stub:

```
function W(value, name) {
    if(stub.write exists)
        value = stub.write(value, name)
    return value;
}
```

When executing the instrumented code, it not only performs the original semantics but also calls those stub function as dynamic programming analysis interfaces. But This is just a very simple case for the ease of understanding. JavaScript is a very flexible programming language with many dynamic programming language features, this requires our framework to be able to instrument many other programming constructs such as object/function/regexp/array literals, condition, loop, method/function call *etc.*

Despite the difficulties of implementing the code instrumentation for JavaScript, the benefits of doing code transformation are many-folds 1) it provides finer granularity of dynamic analysis interface; 2) no modification to the virtual machine is required; 3) analysis code is written in JavaScript which is familiar to front-end developers; 4) the JavaScript analysis framework is highly portable, as long as the JVM adheres to the ECMAScript standard [13], the JavaScript analysis framework can work on it with trivial migration effort.

As we adopt source manipulation to add hooks instead of modifying the underlying virtual machine and consequently the front-end analysis framework fuses analysis code into the programming context of the target code. The analysis code is executed in an external function to be called during the execution of the transformed target code. So the analysis code programming context and paradigm is the same as the JavaScript application. This makes it easy for any JavaScript developer to write their own analysis module. In contrast existing dynamic analysis framework often requires an understanding of the underlying virtual machine or physical machine mechanism, which is error-prone and demands a shift of programming paradigm.

4 Random Testing

Random testing is a black-box software testing technique in which test inputs are generated randomly. In Figure 1, `J$1` and `J$2` in student's solution (SS-1) reads randomly generated inputs and `J$.setOutput()` records the program output under the corresponding input. Same inputs are sent to reference implementation (RI-1 or RI-2) and `J$.setOutput()` records the reference implementation's output as the test oracle to determine whether the test passes or fails.

Although random testing generally works well for most programs, some bugs can hide in branches that may rarely be reached by random testing. For example, in the following listing, if we draw value for variable `a` and `b` randomly from an integer pool ranges from -100000 to 100000, the probability that we generate a test case that enters then branch would be $\frac{1}{200000}$.

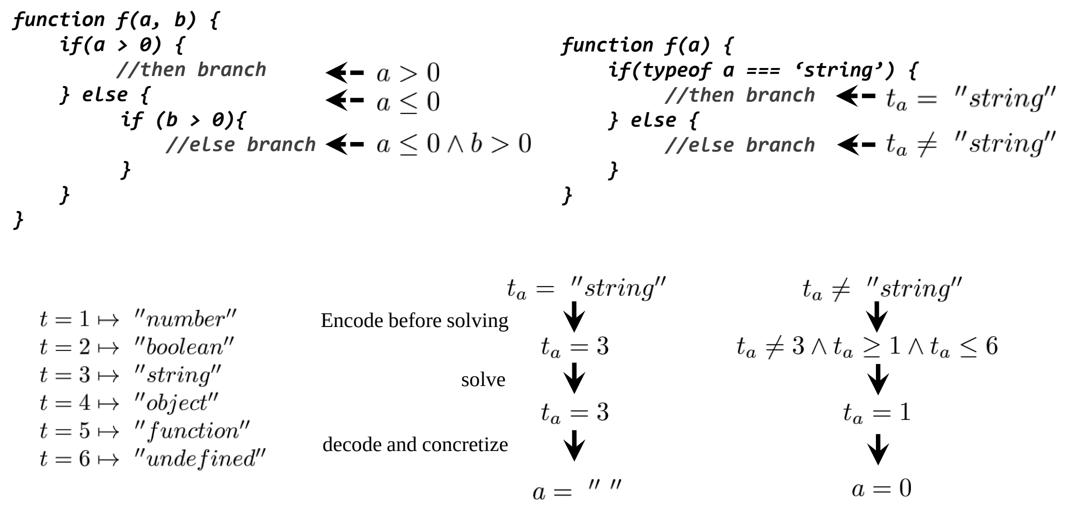
```
if(a == 232) {
    // error !;
}
```

So we need a more advanced technique to help enter those corner cases and branches.

5 Symbolic Execution

Symbolic execution[19, 24] is a kind of abstract interpretation technique that, instead of computing concrete values during the program execution, calculates intermediate values using logic formulas. Traditional static symbolic execution (*abbr.* SSE) interprets the program statically [24]. This usually cannot achieve high coverage, as the program may often turn into higher order logic (*e.g.,* higher order functions are turned into first order logic) or contain entities (*e.g.,* objects) that the SMT solver[11] is incapable of handling. Dynamic symbolic execution (*abbr.* DSE) techniques [8, 20, 38, 7] were proposed to address this issue. DSE executes the program symbolically when the generated formula can be handled by SMT solver, when it encounter entities or operations that leads to SMT solver out of theory, DSE uses concrete values instead. After executing the program once, symbolic execution technique generates a path constraint consisting of all path predicates collected along executing the programs. In order to generate a new test case that covers a previously unseen path. DSE negates the last path constraint and send the mutated path constraint into the SMT solver. The solver solves the path constraint and create a new test case that will lead to a execution covering other parts of the program under test.

To achieve better test coverage than existing system [9], we implement our own dynamic symbolic execution engine for JavaScript programs. Our DSE engine works in a similar way as KLEE [7] except 1) KLEE is implemented on top of LLVM [25] framework while our symbolic execution adopts program instrumentation and dynamic program analysis (section 3) to perform abstract interpretation and thus enables generating test cases; and 2) our implementation handles advanced programming features in JavaScript to achieve higher test coverage.



```
function f(a, b) {
    if(a > 0) {
        //then branch
    } else {
        if (b > 0){
            //else branch
        }
    }
}
```

```
function f(a) {
    if(typeof a === 'string') {
        //then branch
    } else {
        //else branch
    }
}
```

Figure 3: Handle `typeof` Operator in JavaScript.

6 Handle Advanced Features

Conventional symbolic execution are performed on either assembly or low level programming language (C/C++), so there is a limited kind of instructions and statements to handle by the symbolic execution.

However, JavaScript is a dynamic programming language incorporating many new features which further widen the gap between the programming language semantics and the SMT solver's theory.

As an example, Figure 3 shows a simple program (in the left-upper side) with three possible branches. Existing symbolic execution tools will be able to handle this case properly as the path predicate is within the theory of SMT solvers. However, another simple program (in the upper-right side) with only two branches is beyond the capability of the existing symbolic execution engines. As the program uses unary operator `typeof` which returns a string representing the type of the operand. The `typeof` operator could return one of six values: `number`, `string`, `object`, `function`, `undefined` and `boolean`.

In order to handle this case, we extend the symbolic execution engine so that when it reaches the branch conditional expression `typeof a === 'string'` it generates a fresh symbolic variable $typeof\_\_a$ corresponding to the type of variable `a`. The existing SMT solver is capable of generating concrete input leading to the then branch as solving the constraint $typeof\_\_a =' string'$ is within its supported theory, and we next decode the constraint solver output to randomly generate a concrete input of type `string`.

The tricky part is generating the input leading to the `else` branch. As the path constraint turns into $typeof\_\_a \neq' string'$. A SMT solver may generate a random string as the solution which is meaningless. In order to constrain the output of SMT solver, we first decode all typeof symbolic values into integers according to the mapping relation shown in Figure 3 (in the left-down corner). For example, `string` type is mapped to integer 3 and $typeof\_\_a \neq' string'$ is encoded into $typeof\_\_a \neq 3$. In order to constrain the solution of the SMT solver in a meaningful domain, before each SMT query is sent to the SMT sovler, for each typeof constraint variable $typeof\_\_x$ we added an additional constraint: $typeof\_\_x \geq 1 \wedge typeof\_\_x \leq 6$. So now the encoded path constraint turns into:

$$typeof\_\_x \neq 3 \wedge typeof\_\_x \geq 1 \wedge typeof\_\_x \leq 6$$

For this query, the SMT solver generates the solution: $typeof\_\_x = 1$ followed by a post-processing step in our engine which decodes $typeof\_\_x = 1$ into $typeof\_\_x =' number'$ according to the map shown in Figure 3. Our symbolic execution engine then accordingly generates a concrete input of type `number` which guides the concrete execution to the `else` branch to achieve full test coverage.
Auto-grading is handled according to the ratio of test cases that fails. Some auto-grading system considers syntactic similarity as one factor for auto-grading deterministic finite automaton (DFA) assignments [3]. Here we only considers ratio of test cases that fails, as syntactically correct programming implementations to the same problem can be very different from each other. Besides, in our system reference implementation can simply invoke native function calls where source code are not available (*e.g.,* RI-1 in Figure 1).

7 Statistical Bug Localization

Spectrum-based fault localization(or statistical fault localization) techniques find suspicious faulty program entities in programs by comparing passed and failed executions.

Spectrum-based fault localization aims to locate faults by analysing program spectra of passed and failed executions. A program spectra often consists of information about whether a program element (e.g., a function, a statement, or a predicate) is hit in an execution. Program spectra between passed and failed executions are used to compute the suspiciousness score for every element. All elements are then sorted in descending order according to their suspiciousness for developers to investigate. Empirical studies (e.g., [29, 22]) show that such techniques can be effective in guiding developers to locate faults. Parnin *et al.* conduct a user study [32] and show that by using a fault localization tool, developers can complete a task significantly faster than without the tool on simpler code. However, fault localization may be much less useful for inexperienced developers.

Before we start looking at the fault localization, several concepts need to be stated. A **program spectrum**[3] [2] is a collection of data that provides a specific view on the dynamic behavior of software. This data is collected at run-time, and typically consist of a number of counters or flags for the different parts of a program.

$$\begin{array}{cc} M \text{ spectra} & \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1N} \\ x_{21} & x_{22} & \cdots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \cdots & x_{MN} \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} \\ & \begin{matrix} s_1 & s_2 & \cdots & s_N \end{matrix} \end{array}$$

Figure 4: An illustration of Program Spectra.

In figure 4, each line represents a test case execution trace, while each column means a specific program entity(statement, block, or function)'s hit count in all test cases. The fault localization techniques rank each feature in the matrix according to their relevant score to the execution result. It is actually a feature selection on program spectrum.

The key for a spectrum-based fault localization technique is the formula used to calculate suspiciousness. The existing state-of-arts in fault localization fields are Jaccard[26], Tarantula[23] and Ochiai[26] etc. Zhang et al. summarizes 33 existing fault-localization techniques and compare their effectiveness on seimens benchmark dataset[12]. Table 1 lists the formulae of three well-known techniques: *Tarantula* [22], *Ochiai* [1], and *Jaccard* [1]. Given a program element s, $N_{ef}(s)$ is the number of ***failed*** executions that ***execute*** s; $N_{np}(s)$ numerates ***passed*** executions that do ***not*** hit s; by the same token, $N_{nf}(s)$ counts ***failed*** executions that do ***not*** hit s and $N_{ep}(s)$ counts ***passed*** executions that ***execute*** s.

Table 1: Spectrum-based fault localization

Name	Formula
Tarantula	$\dfrac{\frac{N_{ef}(s)}{N_{ef}(s)+N_{nf}(s)}}{\frac{N_{ef}(s)}{N_{ef}(s)+N_{nf}(s)} + \frac{N_{ep}(s)}{N_{ep}(s)+N_{np}(s)}}$
Ochiai	$\dfrac{N_{ef}(s)}{\sqrt{(N_{ef}(s)+N_{nf}(s)) \cdot (N_{ef}(s)+N_{ep}(s))}}$
Jaccard	$\dfrac{N_{ef}(s)}{N_{ef}(s)+N_{nf}(s)+N_{ep}(s)}$

Example. N_{ef}, N_{ep}, N_{nf}, and N_{np} can thus be calculated from the spectra. The suspiciousness scores of *Tarantula*, *Ochiai*, and *Jaccard* for each statement are then calculated based on the formulae in Table 1.

8 Generate Counter Example

Although symbolic execution and random testing generates test cases that cover branches and paths, high coverage does not always guarantee finding a bug triggering test input. Without counter examples, statistical

[3]Figure 4 is from from [2], because their explaination is accurate and better.

bug localization approaches can not pinpoint bugs as there is no behaviour difference between the buggy program and reference implementation. To avoid this potential issue, we adopt a technique for equivalence checking to find counter examples[39]. Specifically, given student solution p_s and reference implementation p_r, we check the semantics equivalence of the two programs by combining with the template:

```
$p_c$ = function (input) {
  out1 $\leftarrow p_s$(input);
  out2 $\leftarrow p_r$(input);
  if (out1 $\neq$ out2) error;
}
```

The combined program p_c will be executed through the symbolic execution engine, if a test case entering the last then branch is feasible, that is an counter example as the outputs of the two programs (p_s and p_r) are different under the same input.

9 Practical Issues

Our system adopts dynamic program analysis based on concrete testing which allows us to collect runtime behaviour information for providing feedback, but also brings some unique issues comparing with statically checking the program. The following sections describe some of the practical issues for building a usable system based on dynamic program analysis and how we deal with them.

9.1 Detect Cheating

Invoking native library as reference implementation can save instructor's effort. However, that also means s-tudent can cheat by invoking those native library. For example, if the programming assignment is implement-ing a sorting algorithm, the student's submission may simply contains a statement calling `array.sort` which is a library method integrated in most JavaScript environment.

To address this issue, our program instrumentation framework rewrite function calling expression to intercept every function or method call to enable additional checking. For this issue, beforing calling a function/method f, our system checks if $f \in S_f$ which is a set of forbidden functions configurable by the instructor.

9.2 Kill Infinite Loops

Student solution may contain any form of bugs, including a loop that may never ends. For concrete testing based approach, that means evaluating the student's solution may leads to a frozen browser. Our system addresses this issue by instrumenting conditional expression in `while` and `for` loops to check if the conditional expression always evaluates to $true$ for certain times. If the count is above a predefined threshold, the dynamic analysis terminates the loop by assigning the conditional expression a $false$ value. A warning on infinite loop also be reported.

9.3 Detect Inefficient Code Patterns

Another key building block of our system is to detect bad code practise and inefficient code in student solution and generate auto-feedback to guide students towards production level code.

Traditional JavaScript code interpreters are slow. As more intensive and important computation are carried out in the browser and node.js[4], the performance of JavaScript program becomes the new era of competition among all major JavaScript engine vendors. In recent years, there has been great performance improvement in JavaScript engines, and browsers like Google Chrome and Mozilla Firefox have arguably some of the fastest JavaScript engines in existence now. And many research has been conducted on improving JavaScript engine performance. Based on these research most modern browsers employ some form of compilation and optimization of JavaScript code into machine code or intermediate representation at runtime. Such kind of compilation is called Just-In-Time (*a.k.a.* JIT) Compilation. JIT compilation helps to significantly increase execution speed of JavaScript programs. Substantial research has been done to improve JavaScript engines performance on either benchmarks or real-world websites. However, not many research is conducted on the other direction: pinpointing and re-factoring inefficient JavaScript code patterns to make them run faster on existing JavaScript engines.

We study how major JavaScript engines works and and identify several inefficient code patterns that may slow down the program. We implemented the JIT-compiler unfriendly code detector which monitors the program execution and detect inefficient patterns at runtime.

Based on some coding guidelines[5] specified by JIT compilers and our study into those JavaScript engines, we list several rules for composing efficient code are:
(1) Initialize all object members in constructor functions (so that the instances do not change type later).
(2) Always initialize object members in the same order.
(3) Make sure that a variable has a single consistent type throughout an execution. This guideline becomes more important if the variable holds an integer value.
(4) Use contiguous keys starting at 0 for arrays.
(5) Do not load uninitialized or deleted elements.
(6) Do not store non-numeric values in numeric arrays.
(7) Monomorphic use of operations is preferred over polymorphic operations.
There are similar guidelines for other JIT compilers. Some of the guidelines such as (2) are applicable to other JIT compilers. We implement those checkers in our dynamic analysis framework to detect any of those inefficient code patterns in the student solution when testing the program.

10 Experimental Results

This sections first briefly introduce the implementation of our tool, and followed by the description of evaluating the correctness and performance of our dynamic analysis framework by testing our framework on several well known benchmarks. The results show that after transformation, the semantics of the target program is preserved, and the slowdown caused by our framework for web page rendering and dynamic visual effect is not obvious.

10.1 Implementations

Our system is purely implemented in JavaScript and running fully in the browser to support online education. The entire system contains over 10K lines of code which implements code editing, code instrumentation, evaluation, testing generation, bug localization and program analysis. All those tasks are carried out in the

[4]Node.js is a platform for server-side JavaScript.
[5]`http://www.html5rocks.com/en/tutorials/speed/v8/`

browser at client side and consequently lightens the use of server which further enables massive number of students using the system simultaneously.

10.2 Benchmarks

We created our benchmark set with problems taken from the Introduction to Data Structure (CS 61B) at Berkeley as well as implementation of classic data structures and algorithms. A brief description of each benchmark program is listed as follows:

- `QSort`: Implement quick sort algorithm that takes an array of integers as input.
- `DLlist`: Implement Double Linked List data structure.
- `BTree`: Implement Binary Search Tree data structure.
- `HSort`: Implement heap sort algorithm that takes an array of integers as input.
- `MSort`: Implement merge sort algorithm that takes an array of integers as input.
- `Max`: Implement max function that returns the maximum value among the inputs.
- `Smoosh`: takes an array of integers and replaces consecutive duplicate numbers.
- `Squish`: takes an array of integers and whenever multiple consecutive items are equivalent, removes duplicate elements so that only one consecutive copy remains.

We search the Internet to find implementations written in JavaScript or if no implementation publicly available we implement them manually. To simulate the use of these programs in the real scenario, we seed bugs according to mistakes commonly made by students [40]. The bug patterns we seeded in the implementations are as follows:

Err-1	`a[i]` →	`a[i+1]`|`a[i-1]`
Err-2	`v=n` →	`v=n+1`|`v=n-1`|`v=0`
Err-3	`v1` *op* `v2` →	`v1+1` *op* `v2` — `v1-1` *op* `v2`
		`v1` *op* `v2+1` — `v1` *op* `v2-1`
Err-4	`return v` →	`return v+1` — `return v-1`
Err-5	`a<b` →	`a<=b` — `a>b` — `a>=b`
Err-6	`a==b` →	`a!=b`
Err-7	`a!=b` →	`a==b`
Err-8	`arrIdx` →	`arridx` — `Arridx` — ...

op ∈ { `+`, `-`, `*`, `/`, `%` }

Figure 5: Bugs seeded in benchmark programs

10.3 Bug Localization Accuracy

In this section we present an empirical evaluation that measure the accuracy of our bug localization feedbacks.

Evaluation Metric. We compare the effectiveness of our bug localization component based on diagnostic cost that calculates the percentage of statements examined to locate a fault, which is commonly used in the literature [22, 1, 26]. The diagnostic cost is defined as follows:

$$Cost = \frac{|\{j \mid f_{T_S}(d_j) \geq f_{T_S}(d_*)\}|}{|\mathcal{D}|}, \tag{1}$$

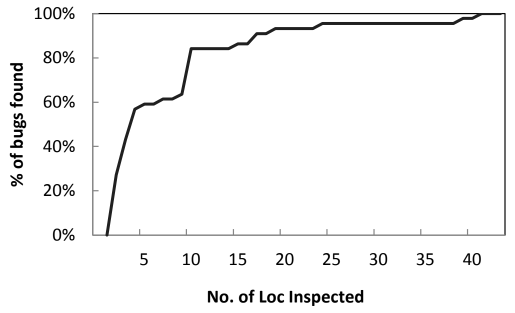
where $\mathcal{D}$ consists of all program elements appearing in the input program. We calculate the cost as the percentage of elements that developers have to examine until the root cause of the failures (d_*) are found. Since multiple program elements can be assigned with the same suspicious score, the numerator is considered as the number of program elements d_j that have larger or equal suspicious scores as that of d_*.

Evaluation Assumption. To measure the effectiveness of interactive fault localization methods with user feedback, we assume that:

1. User inspects program elements following recommended list from the most suspicious to the least.

2. Although our framework provides an option for a user to submit either no or multiple feedback. For the simplicity of evaluation, we assume that a feedback is submitted after each program element is inspected and a user will keep labeling until faults are found.

Benchmark/Error	Err-1	Err-2	Err-3	Err-4	Err-5	Err-6	Err-7	Err-8	Avg(Err)
`Max`	ϕ	14.28%	ϕ	14.28%	14.28%	ϕ	ϕ	14.28%	14.28%
`QSort`	51.11%	4.44%	35.56%	35.56%	6.67%	2.22%	ϕ	2.22%	19.68%
`MSort`	2%	18%	ϕ	ϕ	28%	16%	ϕ	2%	13.2%
`HSort`	60.32%	3.17%	1.59%	ϕ	ϕ	ϕ	14.29%	1.59%	16.19%
`DLlist`	ϕ	1.37%	24.68%	ϕ	5.48%	12.33%	ϕ	8.22%	10.41%
`BTree`	ϕ	1.32%	1.32%	0.66%	1.32%	ϕ	26.49%	1.99%	5.52%
`Smoosh`	56.25%	18.75%	18.75%	ϕ	56.25%	ϕ	56.25%	12.5%	36.46%
`Squish`	56.25%	18.75%	18.75%	ϕ	56.25%	ϕ	56.25%	12.5%	36.46%
Avg(Prgm)	45.19%	10.01%	16.78%	16.83%	24.04%	10.18%	38.31%	6.91%	



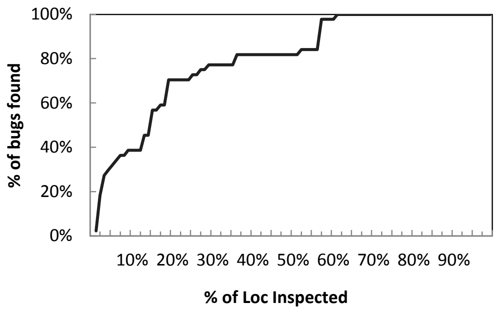


Figure 6: Bug Localization Accuracy.

Evaluation Result. Figure 6 shows all the bugs seeded in benchmark programs. ϕ means the there exists no corresponding programming constructs in the solution to seed the bug. Numbers in the table shows the percent of lines of code to inspect to identify the real bug. For example, `QSort` needs inspecting 2% of the code to find the bug seeded by pattern Err-2 in Figure 5. On the left bottom side of Figure 6, the graph shows the number of code inspected and the corresponding percent of bugs located. For instance, inspecting 5 lines of code on average can locate 60% of bugs, and inspecting 12 lines of code on average can locate over 80% of bugs.

To simulate student solution in the experiment, we search the Internet to use publicly available code snippet. Surprisingly, we find that top 5 search results of ”JavaScript Quick Sort” contain unreported bugs. Those implementations are written for blogs explaining how quick sort works, but the authors only write a few test cases to show the code works without comprehensive testing. Our system detects those bugs by high coverage testing, user only need to simply click “Run”. One of the bugs in Quick Sort from `Github Gist` is easily located by our system and we fix the bug and report it back to the user [6].

10.4 Conformance Test

To make sure that the program instrumentation does not change the semantics of the program. We evaluate our dynamic program analysis framework under conformance test. Test262 is a test suite intended to check agreement between JavaScript implementations and ECMA-262, the ECMAScript Language Specification (currently 5.1 Edition). The test suite contains thousands of individual tests, each of which tests some specific requirements of the ECMAScript Language Specification.

The Firefox SpiderMonkey failed 111 conformance among 1236 conformance test The instrumented SpikderMonkey failed 211 conformance tests. Due to limited time we have, we did not go through those failures one by one but an overview shows that the additional failures are caused by removing the `"use restrict;"` clause from the target code. We remove the clause because it prevents us from using some of the advanced dynamic features of JavaScript (*e.g.,* `arguments.callee`, `eval` *etc.*).

10.5 Performance Test

Our system transforms programs into intermediate representation, and thus perform program profiling and program analysis. In this section, we measure the performance of the dynamic analysis framework on both SunSpider benchmark and our assignment benchmarks.

Testing on the programming assignment benchmarks we collected show that running the entire process(see Figure 2) takes 2s at most on each program.

SunSpider is a benchmark suite that aims to measure JavaScript performance on tasks relevant to the current and near future use of JavaScript in the real world, such as encryption and text manipulation. The suite further attempts to be balanced and statistically sound. The result of SunSpider Benchmarks on our platform is shown in Table 2. On average, the code after transformation is 150X slower than the original target code. The slowdown drastically increases on this benchmark because most of the calculation is performed using the JavaScript code which will be fully transformed. But for most real-world websites, the JavaScript code invokes lower level native library functions provided by the browser (implemented in C/C++) to perform most compute-intensive tasks. And since our framework is only interested in and transforms the top level JavaScript code, the slowdown in this scenario is much less. Although the slowdown on SunSpider seems intimidating, evaluation on the programming assignment benchmarks shows that the entire feedback generating process (including testing, bug localization and program analysis) takes at most 2s for each assignment as the programming assignments are simple algorithms and data structures and thus do not have scalability issues.

[6]`https://gist.github.com/springuper/4182296`

Table 2: SunSpider Benchmark Running on Firefox Before and After Transformation.

TEST	COMPARISON	TIME(TRANSFORMED)	TIME(ORIGINAL)
3d:	140.6x	4582.8ms +/- 2.30%	32.6ms +/- 5.80%
cube:	95.2x	1294.6ms +/- 2.60%	13.6ms +/- 13.90%
morph:	291.7x	1575.2ms +/- 3.80%	5.4ms +/- 12.60%
raytrace:	126.0x	1713.0ms +/- 5.70%	13.6ms +/- 5.00%
access:	333.0x	6060.8ms +/- 2.50%	18.2ms +/- 17.00%
binary-trees:	612.1x	1713.8ms +/- 4.30%	2.8ms +/- 19.90%
fannkuch:	391.2x	2582.0ms +/- 1.60%	6.6ms +/- 10.30%
nbody:	257.7x	1185.4ms +/- 9.90%	4.6ms +/- 52.70%
nsieve:	138.0x	579.6ms +/- 3.70%	4.2ms +/- 13.20%
bitops:	576.0x	5183.6ms +/- 1.70%	9.0ms +/- 25.80%
3bit-bits-in-byte:	1814.0x	1451.2ms +/- 4.60%	0.8ms +/- 69.50%
bits-in-byte:	654.1x	2093.0ms +/- 2.10%	3.2ms +/- 32.50%
bitwise-and:	357.6x	643.6ms +/- 3.70%	1.8ms +/- 30.90%
nsieve-bits:	311.2x	995.8ms +/- 1.10%	3.2ms +/- 17.40%
controlflow:	957.2x	2488.6ms +/- 1.50%	2.6ms +/- 26.20%
recursive:	957.2x	2488.6ms +/- 1.50%	2.6ms +/- 26.20%
crypto:	215.2x	4089.2ms +/- 3.40%	19.0ms +/- 15.30%
aes:	134.0x	1340.4ms +/- 4.10%	10.0ms +/- 21.50%
md5:	289.7x	1332.4ms +/- 4.70%	4.6ms +/- 14.80%
sha1:	321.9x	1416.4ms +/- 5.10%	4.4ms +/- 15.50%
date:	91.3x	2336.4ms +/- 4.10%	25.6ms +/- 12.70%
format-tofte:	127.3x	1730.8ms +/- 3.00%	13.6ms +/- 10.40%
format-xparb:	50.5x	605.6ms +/- 13.50%	12.0ms +/- 19.40%
math:	229.5x	4315.0ms +/- 3.90%	18.8ms +/- 5.50%
cordic:	777.8x	1866.6ms +/- 3.50%	2.4ms +/- 28.40%
partial-sums:	60.2x	866.4ms +/- 4.10%	14.4ms +/- 4.70%
spectral-norm:	791.0x	1582.0ms +/- 8.10%	2.0ms +/- 0.00%
regexp:	1.13x	23.4ms +/- 2.90%	20.8ms +/- 2.70%
dna:	1.13x	23.4ms +/- 2.90%	20.8ms +/- 2.70%
string:	66.3x	5000.2ms +/- 2.80%	75.4ms +/- 22.20%
base64:	80.0x	735.8ms +/- 4.10%	9.2ms +/- 6.00%
fasta:	175.3x	1367.0ms +/- 3.60%	7.8ms +/- 17.50%
tagcloud:	50.1x	861.0ms +/- 6.40%	17.2ms +/- 27.70%
unpack-code:	36.8x	772.8ms +/- 11.90%	21.0ms +/- 16.70%
validate-input:	62.6x	1263.6ms +/- 3.90%	20.2ms +/- 68.20%
TOTAL	**153.5x**	**34080.0ms +/- 1.70%**	**222.0ms +/- 5.90%**

11 Related Work

`FindBugs`[21] is a static code checker that detects error-prone code patterns. In comparison, our analysis framework allows user defined analysis module to detect dynamic operation patterns. Our dynamic analysis framework also allows building other checkers that can help improve static checkers. For example, `JSLint` is a JavaScript static code checker that detects the evil use of JavaScript programming language[10]. In which it deprecates the use of function `eval` considering performance issues and the unpredictable behaviour. `JSLint` can only detect the explicit use of `eval` due to the limitation of alias analysis, and the detection might be unsound as the call expression might never be executed. Our analysis framework on the other hand is capable of accurately detecting both explicit and implicit use of `eval` function and thus can be used as a substitution or supplement to those static checkers.

Many static and dynamic analysis tools [33, 43, 4, 18] for JavaScript have been proposed. Richards *et. al.* did an empirical study[35] on the use of dynamic behaviours of JavaScript programming language in real-world websites. Their results show that most type-checking techniques for JavaScript are based on some strong assumption which are rarely hold in real-world web applications. Newsome *et. al.* implemented a dynamic taint analysis framework[31] for security analysis in web browser. Feldthaus *et. al.* proposed an automatic refactoring tool[17] to avoid using the evil parts of JavaScript code.

Our JavaScript dynamic analysis framework is build on top of `Jalangi`[36, 37] which is a dynamic analysis framework mainly for back-end JavaScript on `Node.js`. But Vast majority of JavaScript code is written for front-end running in web browsers and before `Jalangi` and `Jalangi` does not fully support front-end JavaScript engine like Firefox. Before our framework, there exists no dynamic analysis framework for front-end in browser analysis for JavaScript similar to `Valgrind`[30], `PIN`[27] or `DynamoRIO`[6] for x86. So we design and implement this in-browser framework for front-end JavaScript dynamic analysis. Based on `Jalangi`, we re-factored and rewrote some critical components of the original framework to make it fully support and easily analysing any real-world web pages. Further more, our front-end analysis framework completely decoupled from the JVM and can seamlessly work with web console and debugger in the browser which reuses the UI provided by Firefox. This facilitates developers more intuitive interaction such as analysis module plug in and remove on the fly, and thus makes our framework even more powerful.

12 Conclusion

In this paper we present a system for automatically providing feedback for introductory programming assignments for JavaScript. The system is a test-based approach that can complement formal-method-based approaches and provides more comprehensive testing and more utilities against existing test-based auto-grading systems. The technique uses symbolic execution, statistical bug localization, program analysis to automatically grade, recommend bug locations, and generate program analysis report to students. We have evaluated our technique on a set of benchmarks we collect and the results show that inspecting 5 lines of code recommended by our system, 60% of bugs common made by students can be located. We believe this technique can be a useful platform for not only massive online education courses, but also code interviewing and JavaScript development.

A demo of the Auto-grading system is available at:

`https://www.eecs.berkeley.edu/~gongliang13/jalangi_ff/fault_loc.htm`

References

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 2009.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *TAICPART-MUTATION*, 2007.

[3] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of dfa constructions. In F. Rossi, editor, *IJCAI*. IJCAI/AAAI, 2013.

[4] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In A. P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 428–452. Springer, 2005.

[5] H.-J. Boehm and C. Flanagan, editors. *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 2013.

[6] D. Bruening, T. Garnett, and S. P. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO*, pages 265–275. IEEE Computer Society, 2003.

[7] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In R. Draves and R. van Renesse, editors, *OSDI*, pages 209–224. USENIX Association, 2008.

[8] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[9] Codility. *https://codility.com/train/*.

[10] D. Crockford. *JavaScript - the good parts: unearthing the excellence in JavaScript*. O'Reilly, 2008.

[11] L. M. de Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[12] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[13] S. ECMA-262". *http://www.ecma-international.org/publications/standards/Ecma-262.htm*.

[14] G. V. Engine. *http://code.google.com/p/v8/*.

[15] N. J. Engine. *http://www.webkit.org/projects/javascript/index.html*.

[16] N. S. S. J. Engine. *http://nodejs.org/*.

[17] A. Feldthaus, T. D. Millstein, A. Møller, M. Schäfer, and F. Tip. Refactoring towards the good parts of javascript. In *OOPSLA Companion*, 2011.

[18] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for javascript ide services. In *ICSE*, 2013.

[19] P. Godefroid. Test generation using symbolic execution. In D. D'Souza, T. Kavitha, and J. Radhakrishnan, editors, *FSTTCS*, volume 18 of *LIPIcs*, pages 24–33. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[20] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.

[21] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[22] J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, 2005.

[23] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *ASE*, pages 273–282. ACM, 2005.

[24] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[25] C. Lattner and V. S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[26] Lucia, D. Lo, L. Jiang, and A. Budi. Comprehensive evaluation of association measures for fault localization. In *ICSM*, 2010.

[27] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[28] B. Meyer, L. Baresi, and M. Mezini, editors. *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 2013.

[29] P. A. Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound boolean predicates. In *ISSTA*, pages 5–15, 2007.

[30] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[31] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature-generation of exploits on commodity software. In *NDSS*, 2005.

[32] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209, 2011.

[33] B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *PLDI*, 2012.

[34] M. Rhino. *https://developer.mozilla.org/en-US/docs/Rhino*.

[35] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *PLDI*, 2010.

[36] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In Meyer et al. [28], pages 488–498.

[37] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of javascript. In Meyer et al. [28], pages 615–618.

[38] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005.

[39] C. H. Shuvendu Lahiri, Rohit Sinha. Automatic rootcausing for program equivalence failures. *MSR Tech Report*, 2014.

[40] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In Boehm and Flanagan [5], pages 15–26.

[41] M. SpiderMonkey. *https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey*.

[42] M. TraceMonkey. *https://wiki.mozilla.org/JavaScript:TraceMonkey*.

[43] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *POPL*, 2007.

Autograding and Exercise Generation for Visual Finite State Machines (FSMs)

Zachary MacHardy and Jonathan McKinsey

May 16, 2014

Abstract

In the context of Massively Open Online Courses (MOOCs), the ratio of instructor time to student population is drastically and fundamentally different than that of traditional in-person classrooms. To that end, there is a need for automated tools which can aid an instructor in efficiently distributing effort across large student populations, in a way that would otherwise require an enormous amount of human labor. In this report we provide a scalable, instructor-manageable means of producing and checking a large number of student exercises, based on Chapter 5 of Lee and Seshia's *Introduction to Embedded Systems* textbook. By relying on instructors only to produce a small number of archetypical reference exercises from which a large number of unique problems might be produced, we believe that the tool allows a scalable approach appropriate for a either a classroom or a MOOC context.

1 Introduction and Motivation

1.1 Introduction to Embedded Systems

UC Berkeleys EE149: Introduction to Embedded Systems course teaches students the design and analysis of computational systems that interact with the physical world. This course emphasizes practical design in high confidence systems with real-time and concurrent behaviors under the constraints of software components and physical dynamics. The course is broken up into three main components: modeling, design and analysis. Modeling is imitation of a system to identify what a system does. Design is creation of artifacts within the system to discover how a system does its work. Finally, analysis is the dissection of a system to understand why a system does its role.

These cyber-physical systems are represented using UC Berkeley's Ptolemy II, an open-source Java software framework, and Vergil, its graphical interface, as actor-oriented models. An actor-oriented model is a hierarchical interconnection of actors, where actors are software components that execute concurrently and communicate through interconnected ports.

1.2 Massive Open Online Courses

Because of the advent of Massive Open Online Course (MOOC) platforms such as edX, Udacity and Coursera, curriculum normally reserved for a limited number of local students can now be distributed online in a virtually unlimited scale. UC Berkeley has already launched several MOOC instances of its curriculum and EE149 is one of the newest additions to the course lineup.

However, traditional course management and resource allocation do not scale with the massive and geographically-scattered student population of MOOCs. To bring EE149 to the online community, there are two major factors to consider. First, student assignments need to be automatically graded to provide students with prompt meaningful feedback. Autograders reduce the staff workload, scale constantly despite enrollment fluctuations and grade more consistently than human graders. Second, there needs to be a mechanism for automatically generating problem variations in a large enrollment course especially to deter cheating. Problem variations can also be used as extended practice exercises. Sadigh et al. proposes a template-based approach by generalizing and categorizing existing problems into templates where the differentiating elements are parameters [4]. New problems can be generated by selecting different permutations of the parameters within some space of values appropriate to the template. To help ensure the generated problems are of similar difficulty, the problems are created through a bounded number of mutations from the existing problem.

2 Problem Definition

The scope of this project is to address automatic feedback and exercise generation for *Chapter 5: Composition of State Machines* in Lee and Seshia's *Introduction to Embedded Systems* textbook [3] used in EE149. Before proceeding, it would be useful to review the concepts utilized in Chapter 5. A state machine is a system model whose behavior is described using a set of states and transitions between those states [2]. A finite state machine (FSM) is a state machine in which the set of possible states and transitions is finite. A state transition is bound by a guard expression that evaluates to true when a transition should be taken. Each transition can be associated with a set of actions, which involve assignment to either an internal state variable or an output port.

There are two forms of composition of state machines: concurrent composition and hierarchical composition. Concurrently composed state machines are further categorized as side-by-side or cascade, both of which can either be synchronous or asynchronous. In side-by-side composition, inputs and outputs of the state machines are disjoint as there is no communication between the state machines. However, these state machines may share variables for such purposes as modeling interrupts and threads. Cascade, or serial, composition occurs when the output of one state machine is fed as input into another state machine. In hierarchical state machines, a state may have a refinement, or submodel, that is another state machine. The behavior of this internal state machine when entering or leaving the parent state is constrained by the transitions taken to and from the parent state. If reached by means of a special 'history' transition, introduced in Chapter 5, the state of the internal machine is maintained even while not active. It is otherwise reset to an initial state when revisited. The 'preemptive' transition, also introduced in Chapter 5, designates that transition actions of the submodel be ignored when leaving the parent state. In all other cases the transition actions of the submodel are evaluated and performed before transitions in the parent model, though they logically occur concurrently.

Though the chapter introduces the notion of asynchronous and synchronous composition, it does not deal directly with the particulars thereof. For our purposes, then, we are interested in the automatic grading and generation of problems which leverage synchronous concurrent and hierarchical state machines.

3 Approach

3.1 Autograding

The majority of exercises in Chapter 5 involve flattening composite or hierarchical state machines into a single FSM. For the purposes of grading, a solution must be able to leverage visual FSM models constructed by students and compare them to a reference model provided by the instructor. Though it might be tempting to construct an instructor 'solution' for direct comparison, multiple FSM implementations can provide identical, correct behavior to satisfy a given problem. For that reason, a more flexible approach that alleviates the need for an instructor to generate a solution and leverages the problem directly would be preferable. Such an approach would require only the minimal intervention from the instructor, beyond the initial construction of the reference hierarchical or composite state machine. In the pursuit of that flexibility and the automated evaluation of visually created graphical models, we have chosen to utilize an intermediary conversion from the initial model into a format compatible with the NuSMV symbolic model verification tool [1]. Our general approach to automated grading is outlined below.

First, a student is provided with a problem definition detailing a composite or hierarchical state machine, which can be generated either manually by an instructor in Ptolemy II or automatically (See Section 3.2 **Exercise Generation**). A student proceeds to generate a flattened Ptolemy II model intending to encapsulate the same behavior as the instructor provided solution, then the student submits that solution to the autograding software. The software, built on the Ptolemy II source code, takes the provided student solution and the reference instructor solution as input, and performs a translation step, turning each model into a syntactically and semantically correct NuSMV specification. Next, the software takes the two translated model specifications, performs an additional decoration and integration step to combine each specification into one concurrently composed state machine, resolving issues such as namespace overlap. Then the software detects the output of each of the two concurrent models and constructs an Linear Temporal Logic (LTL) specification which requires that each machine's output remain the same for identical inputs. Finally, the NuSMV program is invoked with the composite model and LTL specifications obtained in the previous steps as input. The result is either a counterexample trace proving that the two models diverge in their behavior, which is passed back to the student, or a confirmation of the LTL specification's correctness. In the former case, the student can edit their state machine and attempt the question again. In the latter case, the student solution was correct, and the student can move onto other problems.

There are, however, a number of limitations in our prototype autograder. The autograder, as described, can only verify or disprove the correctness of a student solution and does not provide specific feedback beyond the counterexample trace. Ideally a more nuanced approach would be taken, generating feedback more immediately useful to the student user. Further, due to limitations of the current translation to NuSMV format, the autograder currently only functions for models using boolean valued guard and output expressions. Next steps in the implementation would involve broadening the capability of the translation to include more general state machines, to which no barrier exists either in Ptolemy II or NuSMV.

3.2 Exercise Generation

While the automatic evaluation of exercises is useful in the context of a course the scale of the typical MOOC, it does not address the issue of generating a variety of exercises in the first place, either for the purposes of discouraging cheating or providing extra practice. To that end, we have designed a visual templating approach to exercise generation. Much as previous templating solutions to exercise generation in subjects such as algebra [5] involve manually authoring a general framework for an exercise, which is then populated with operators and operands, our visual templating solution allows an instructor to specify the general form of an exercise, which is then varied in a constrained way to produce a number of new and unique exercises. Our solution to this problem is meant to allow instructors to construct a problem of a general form about which they would like to test students (such as a hierarchical state machine with one refinement), and then produce variations on that same form.

As stated above, our approach first requires that an instructor generates a Ptolemy II model meant to be flattened by a student. This machine can involve refinements, history or preemptive transitions, or composition, depending on what the instructor would like to test. Next, the instructor invokes the problem generation tool, passing as an argument the Ptolemy II XML model. A simple XML parser reads the model specification, associating nodes with their outgoing and incoming transitions, as well as refinements with their parent states. Next, the program varies incoming and outgoing transitions, ensuring that the final machine is still deterministic (no two outgoing edges have the same guard expression) and a syntactically correct description of a Ptolemy II machine. Notably, the generated machine has the same structure as the original archetype, but the guard expressions, output actions, and transition types (i.e. history, preemptive) are non-deterministically varied. Next, utilizing the autograding framework detailed above (Section 3.1 **Autograding**), the generated model is tested to ensure it is not a tautology, a contradiction, or identical to the original model. This process can be repeated to generate as many or as few exercises as an instructor or student requires.

There are some constraints placed on this generation. First, no transitions will be created or destroyed in the parameterization process. The structure of the instructor-provided model will remain static, but the nature of the transitions between states are changed. Additionally, the tool assumes that the set of transitions are exhaustively and deterministically defined in the reference model. Additionally, due to a dependency on the autograder's capability, instructor models are also subject to the same constraints detailed in Section 3.1 **Autograding** .

4 Results

We have constructed and tested a prototype of the autograding and exercise generation tool, based on the Ptolemy II source code. Below are detailed two examples. First, a description the process of grading a composed state machine taken from Lee and Seshia's *Introduction to Embedded Systems* textbook, problem 5.4. Next, there follows an example of a templated exercise based on the same problem, as well as *Introduction to Embedded Systems* 's problem 5.5.

4.1 Autograding Example

First, an examination of the autograding process. As described, the instructor generates a problem (Figure 1) that is a legal Ptolemy II model. A student takes this model and attempts to create an equivalent model from a single state machine. Included for reference is one incorrect solution (Figure 2) with one guard transition incorrectly configured, and a correct solution (Figure 3).

Next, both the reference and student supplied models are converted into an equivalent NuSMV format, in order to be combined by the tool for verification (Figure 4).

Finally, if the model is incorrect, a counterexample trace is given, though this can be hard to understand for a user unfamiliar with the tool (Figure 5). Otherwise, the solution is correct and the student is informed through a simple textual message.

4.2 Exercise Generation Example

As detailed above, the exercise generation process requires the construction of an archetypical 'template' model. Below are simple examples of a templated problem (left), and an automatically generated variation (right). This includes both problem 5.4 (Figure 6) and problem 5.5 (Figure 7) from the Lee and Seshia text.

5 Conclusions and Future Work

5.1 Conclusions

In this technical report we have outlined a process for the automated evaluation and generation of exercises similar to those found in Chapter 5 of the Lee and Seshia *Introduction to Embedded Systems* textbook. In so doing we have implemented a prototype capable of checking and producing models under a set of reasonable constraints, outlined above.

The autograding is accomplished by means of translating reference and student Ptolemy II models into an equivalent NuSMV-compatible symbolic model verification formula, then synthesizing those specifications into one larger specification, which is checked by the NuSMV software for equivalence. Our approach automated grading allows an instructor to avoid generating both a solution and a problem generation for every statement, as student solutions can be checked against the original hierarchical or composite FSM model. Further, though the output is currently difficult for students to parse, we do provide feedback in the form of a counterexample trace, which may be useful for students when trying to locate a mistake.

The exercise generation is performed by modifying the XML description of an instructor generated archetype. By parsing and modifying the XML with a small set of restriction, we are able to produce well-formed and non-trivial variations upon the exercises in the Lee and Seshia text. These models are, however, bound by a set of restrictions, such as boolean valued guards and outputs, detailed above.

The intended purpose of this tool is the generation and evaluation of problems in a MOOC. To that end, it was essential that our solution provide a scalable, instructor-manageable means of producing and checking a large number of student exercises. By relying on instructors only to produce a small number of archetypical reference exercises from which a large number of unique problems might be produced, we believe that the tool allows a scalable approach appropriate for a either a classroom or a MOOC context.

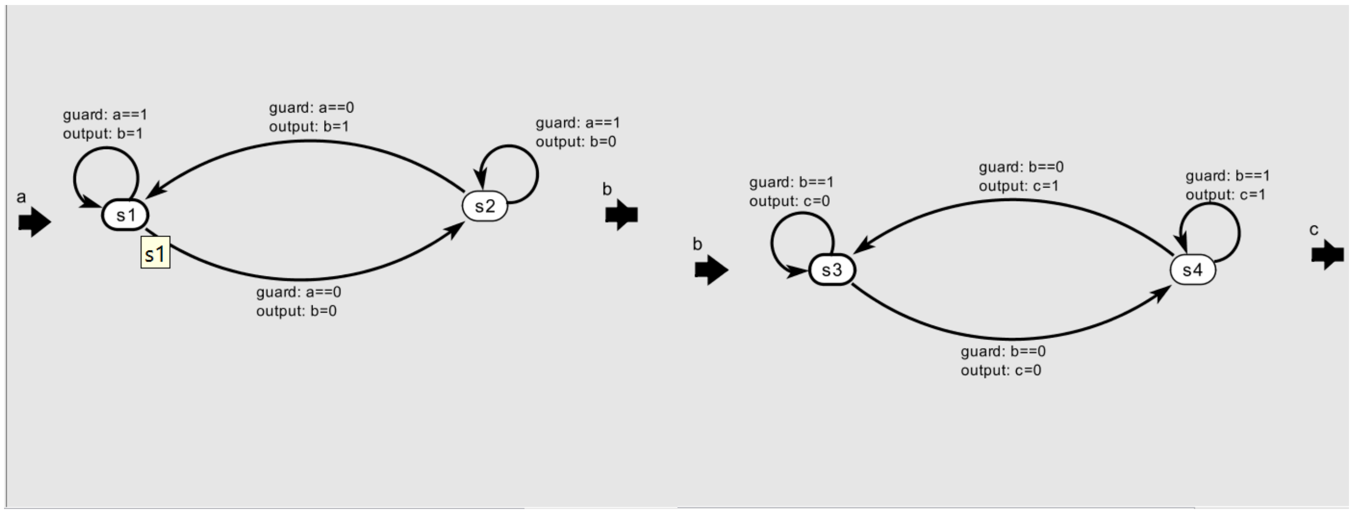


Figure 1: The description of a problem generated by an instructor

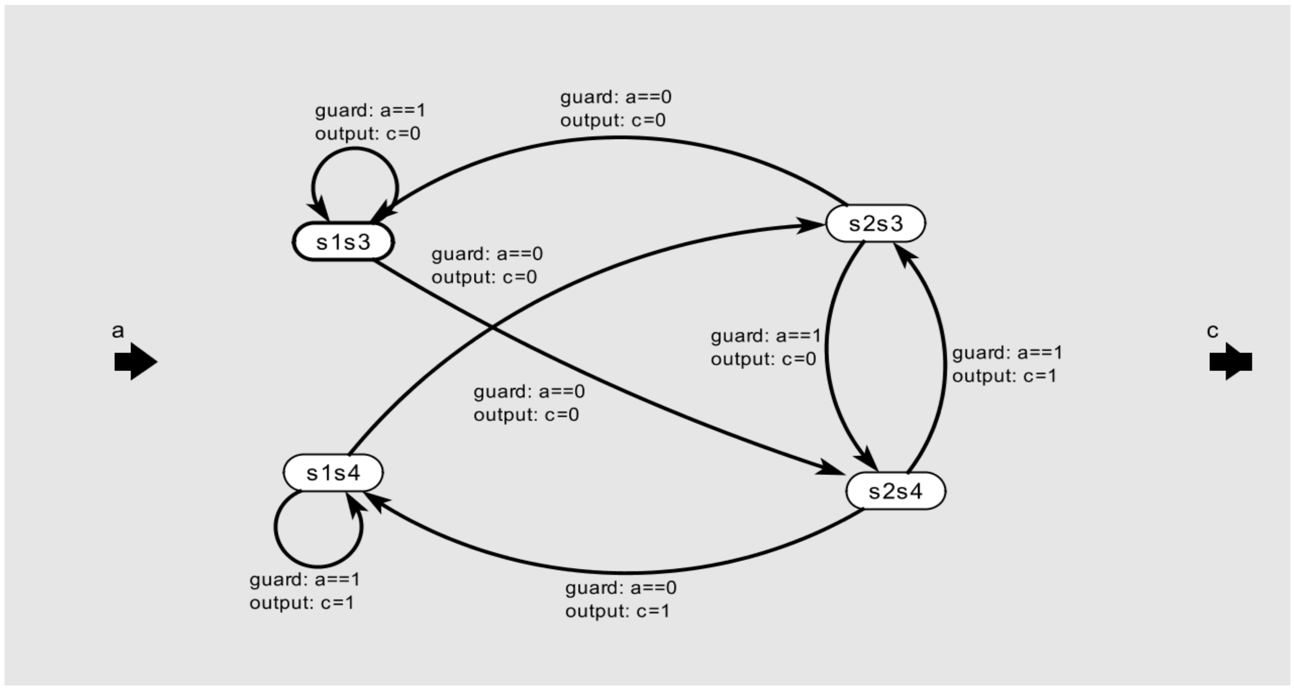


Figure 2: An incorrect student solution

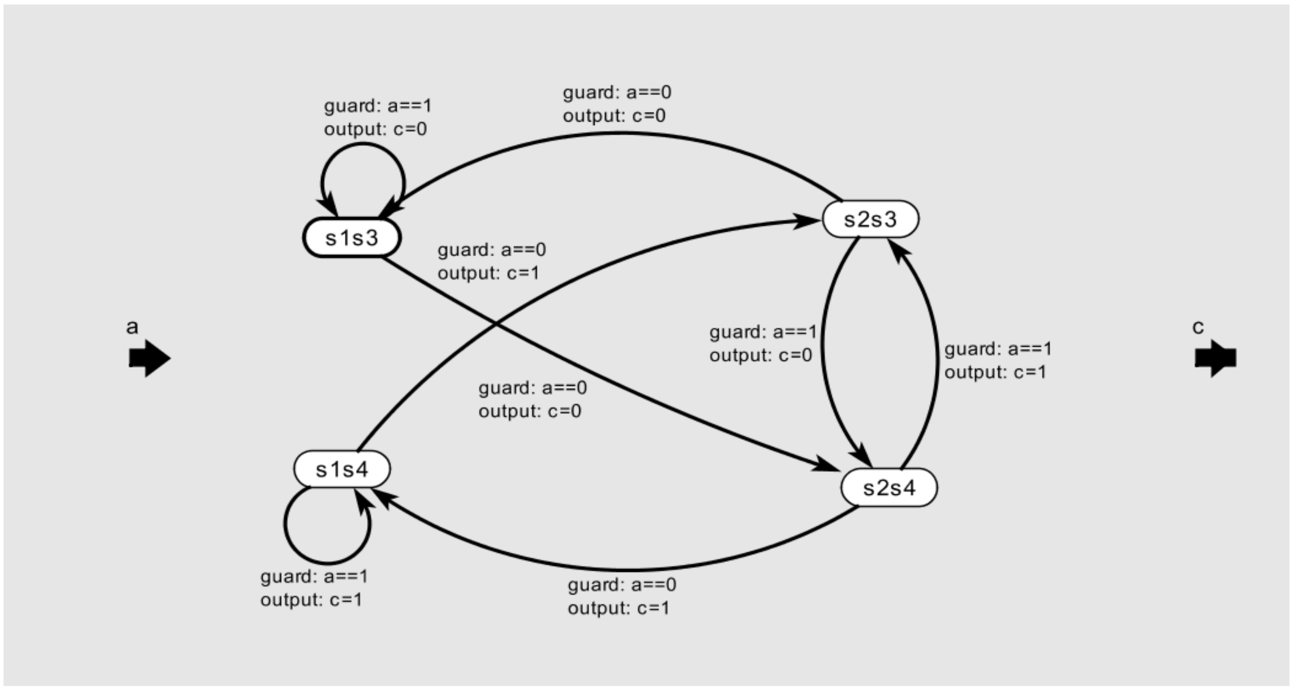


Figure 3: A correct student solution

```
MODULE ModalModel(a_value,a_isPresent)

        VAR
                ModelState : {s2s3,s1s4,s2s4,s1s3};
                a : { ls,0,1,gt };
        ASSIGN
                init(ModelState) := s1s3;
                next(ModelState) :=
                        case
                                ModelState=s1s3 & a_value=TRUE & a_isPresent  :{ s1s3 };
                                ModelState=s1s3 & a_value=FALSE & a_isPresent  :{ s2s4 };
                                ModelState=s2s4 & a_value=TRUE & a_isPresent  :{ s2s3 };
                                ModelState=s2s4 & a_value=FALSE & a_isPresent  :{ s1s4 };
                                ModelState=s2s3 & a_value=TRUE & a_isPresent  :{ s2s4 };
                                ModelState=s2s3 & a_value=FALSE & a_isPresent  :{ s1s3 };
                                ModelState=s1s3 & a_value=TRUE & a_isPresent  :{ s1s3 };
                                ModelState=s1s3 & a_value=FALSE & a_isPresent  :{ s2s4 };
                                ModelState=s1s4 & a_value=TRUE & a_isPresent  :{ s1s4 };
                                ModelState=s1s4 & a_value=FALSE & a_isPresent  :{ s2s3 };
                                TRUE                 : ModelState;
                        esac;
```

Figure 4: The specification generated by the software

```
    compositeModule.Module_B.c_isPresent = TRUE
    compositeModule__Two__.c_isPresent = TRUE
    compositeModule__Two__.ModalModel.c_isPresent = TRUE
  -> State: 1.3 <-
    compositeModule.Module_A.ModelState = s2
    compositeModule.Module_B.ModelState = s4
    compositeModule__Two__.ModalModel.ModelState = s2s4
    compositeModule.c_value = TRUE
    compositeModule.Module_A.b_value = TRUE
    compositeModule.Module_B.c_value = TRUE
    compositeModule__Two__.c_value = TRUE
    compositeModule__Two__.ModalModel.c_value = TRUE
  -> State: 1.4 <-
    compositeModule.Module_A.ModelState = s1
    compositeModule__Two__.ModalModel.ModelState = s1s4
    compositeModule.Module_A.b_value = FALSE
    compositeModule__Two__.c_value = FALSE
    compositeModule__Two__.ModalModel.c_value = FALSE
  -> State: 1.5 <-
    a_isPresent = FALSE
    compositeModule.Module_A.ModelState = s2
    compositeModule.Module_B.ModelState = s3
    compositeModule__Two__.ModalModel.ModelState = s2s3
    match = FALSE
    compositeModule.c_value = FALSE
    compositeModule.c_isPresent = FALSE
    compositeModule.Module_A.b_isPresent = FALSE
    compositeModule.Module_B.c_value = FALSE
    compositeModule.Module_B.c_isPresent = FALSE
    compositeModule__Two__.c_isPresent = FALSE
    compositeModule__Two__.ModalModel.c_isPresent = FALSE
```

Figure 5: A partial trace detailing a counterexample to the student solution

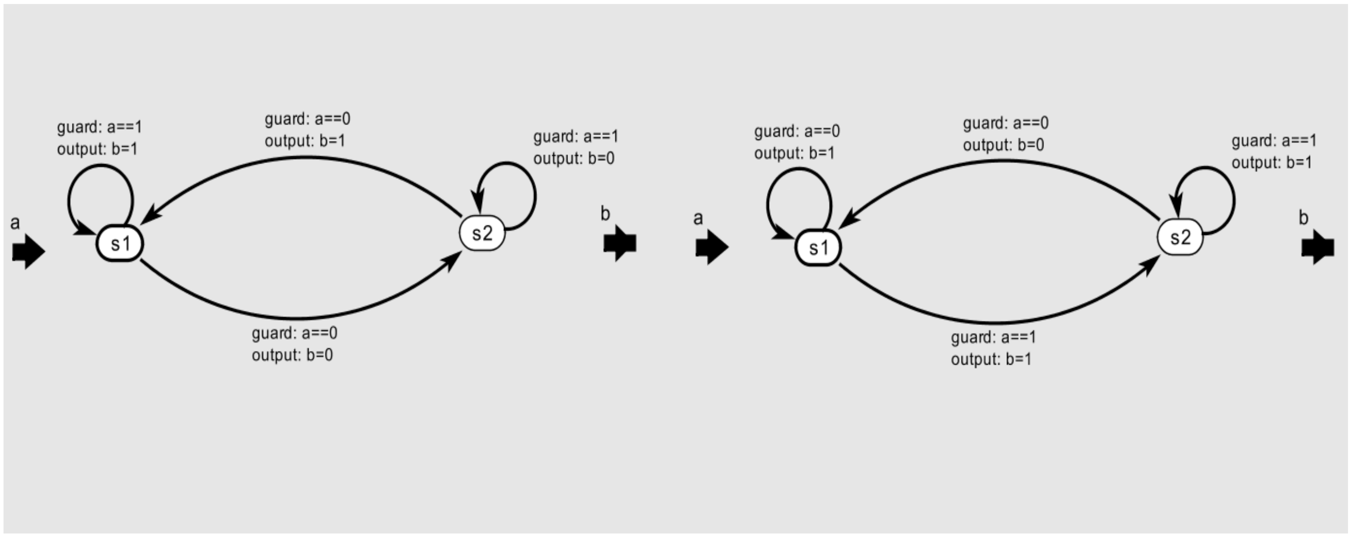


Figure 6: Lee and Seshia problem 5.4 (left) and a generated exercise (right)

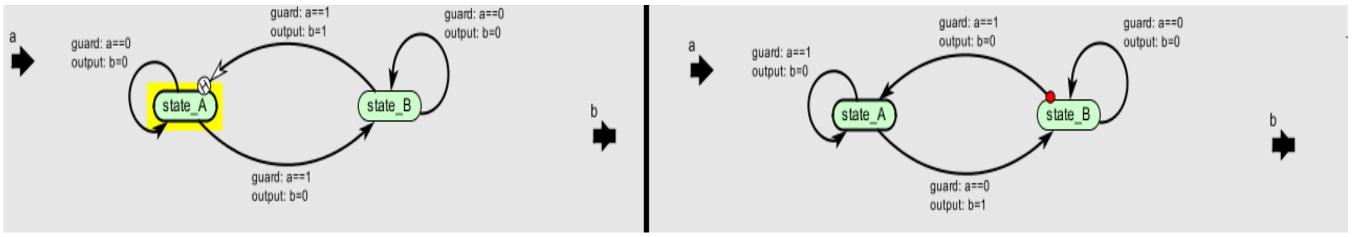


Figure 7: Lee and Seshia problem 5.5 (left) and a generated exercise (right)

5.2 Future Work

There remains many avenues by which the prototype might be improved for implementation in a course. First, an enhancement of the translation and problem-generation portions of the project, to allow for the use of non-boolean guard and output values, would greatly generalize the variety of model that the tool can produce and verify. This might open the door for the tool to be useful for problems outside of the fairly constrained set of exercises in the scope of this report.

Second, the constraints on exercise generation might be refined or customized. It is reasonable that an instructor might prefer certain types of generated exercises over others. Though not currently supported, there is ample opportunity to allow instructors to specify logical constraints on the types of exercises produced, allowing for pruning beyond the removal of trivialities and repeats. Further, the parameterization of templated exercises rely on arbitrarily specified set of probabilities, which determine when guard expressions or transition types are altered. It would be an interesting investigation to determine if one could design an 'instructor-trained' problem generator, which varied these probabilities according to previously accepted or rejected exercises.

Additional work might involve the generation or pruning of states and transitions within a model. For simplicity, the prototype relies on the structure of the instructor-generated model. Due to constraints on the Ptolemy II specification, this involves additional problems of optimized graph placement, and so was not in the scope of the current project. However, it would be useful to be able to generate a more varied set of exercises, not entirely dependant on the structure of the archetype.

References

[1] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, number 2404 in Lecture Notes in Computer Science, pages 359–364. Springer Berlin Heidelberg, January 2002.

[2] Edward A. Lee. Finite state machines and modal models in ptolemy II. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, November 2009.

[3] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. Lee and Seshia, 1 edition, 2010.

[4] Dorsa Sadigh, Sanjit A. Seshia, and Mona Gupta. Automating exercise generation: A step towards meeting the MOOC challenge for embedded systems. In *Proc. Workshop on Embedded Systems Education (WESE)*, October 2012.

[5] Rohit Singh, Sumit Gulwani, and Sriram K. Rajamani. Automatically generating algebra problems. In *AAAI*, 2012.

Assistant for Learning Introductory Python Program

Phitchaya Mangpo Phothilimthana and Cuong Nguyen

May 16, 2014

Abstract

We present a novel solution to the problem of automated feedback generation for introductory Python programs in a massive open online course (MOOC) environment. Compared with the state-of-the-art approaches in the literature, our approach does not require an error model as input, and can handle arbitrary mistakes the students make. We evaluate our approach on a set of benchmarks extracted from previous related work, and from the introductory Python program course CS61A at UC Berkeley. Our evaluation shows that our approach correctly fix the students' buggy solutions, and is within the responsiveness threshold to be useful in a MOOC environment.

1 Introduction

In this paper, we present an automated technique to assist students in learning introductory Python program. Compared to the state-of-the-art Python automatic feedback generator [4], our system is similar in the sense that we depend on the TA solution as the specification, and check whether the student solution is correct by matching all input-ouput pairs. Unlike [4], our approach does not require the TA to write an error model, and can handle arbitrary mistakes the students make. The approache leverages techniques in bug localization and synthesis to automatically determine the bug locations and infer a set of fixes for the programs. To obviate the burden of writing an error model, our system defines several heuristics and generic error model on its own. As a consequence, the search space for synthesis is larger than [4]. We then need a good bug localization method and a better search strategy to prune the search space and quickly find the correct fix. Our contributions on this work is as follow:

- We implemented a complete system for automatic feedback generation for introductory Python program without using a manually written error model.
- Preliminary evaluation on a set of benchmarks extracted from [4] and the course CS61A from UC Berkeley shows that our system is within the responsiveness threshold[1]— the median of time to generate feedback is 7.5 seconds— to be useful in a MOOC environment.

The rest of the paper is arranged as follows. In Section 2 we give an overview on our approach and the architecture of the system. We then discuss three core components of the system, the *Python to Racket Translator*, the *Bug Localization Component* and, the *Feedback Generation Component* in Section 3, 4 and 5 respectively. Section 6 concludes.

[1]As suggested by Nielsen in [3], below ten-second limit can keep a user focus on his or her dialog with the system.

2 Overview

We begin with a brief overview on the system architecture. As depicted in Figure 1, input to our system is the student buggy solution[2], written in Python, and the TA correct solution as a specification, also written in Python. Output of our system is a feedback that helps the student fix the bugs. The feedback tells exactly which expressions are buggy and how to correct them, so that the student solution and TA solution match on all input-output pairs.

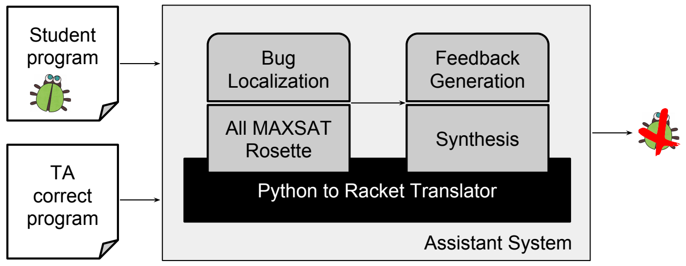


Figure 1: Python Assistant System Architecture

Our system consists of two core components. The first component is a *bug localization system*, which automatically identifies the locations (e.g. expressions, statements, operators, etc.) of bugs. The second component is a *feedback generation system*, which automatically corrects the given buggy expressions and statements. We locate bugs and synthesize correct programs using Rosette [7, 8], a solver-aided language. In order to use Rosette, we implement *Python to Racket translator* to compile students' and TA's Python programs to Racket programs such that Rosette can reason about.

3 Python to Racket Translator

Python and Racket language share many similarities; they are both dynamic language, and many Python language constructs used in introductory courses are also available in Racket, such as if, for loop, while loop, and list. The core differences between Racket and Python are that Python uses infix syntax while Racket use postfix syntax, and that Racket does not support return construct.

To translate Python to Racket, we use the rules specified in Table 1, which shows the corresponding Racket expression or statement for a Python expression or statement. We implemented our source to source translator via syntax-directed-translation using Python AST library. We maintain a map between Racket expression or statement to Python AST node using a pair of line number and column offset.

Some Python constructs are more challenging to translate into a Racket program that is suitable for Rosette's SMT solver, including for and while loop, recursive function and list. We re-define for and while loop in Racket so that they receives an extra parameter which specifies the bound that the loop is un-rolled, which makes the search space finite. Similarly, we add another argument to recursive functions to bound the recursive depth. Finally, for list, Python list allows both appending and indexing operation, while Racket list only allows appending operation, and Racket vector only

[2]If the student solution does not have bugs, the system will feedback that the solution is correct.

Expression or Statement	Python Syntax	Racket Syntax
Variable reference	`var`	`var`
Constant literal	`n`	`n`
Conditional	`if exp01 exp02 else exp03`	`(if exp01 exp02 exp03)`
Assignment	`var = exp`	`(set!  var exp)`
Definition	`var = exp`	`(define var exp)`
Procedural	`proc(exp, ...)`	`(proc exp ...)`
Binary expression	`a op b`	`(op a b)`

Table 1: Python to Racket source to source.

allows indexing operation. Therefore, we use both list and vector in Racket to simulate Python's list.

4 Bug Localization

In order to make the feedback generation system synthesizes the fixes within a reasonable amount of time, the bug localization system must output a set of possible bug locations that is as small as possible.

In this project, we apply two existing techniques to perform bug localizations. The two techniques we choose are both SAT-based. This is because we have already obtained SMT formula of the students and teachers programs by translating the programs implemented in Python into Racket programs, which are in turn translated into SMT formulas by Rosette. We described the two SAT-based bug localizations in the next two sections and present the results in the section after.

4.1 Minimal Unsatisfiability Core

One technique of bug localizations is using minimal unsatisfiable core (minimal UNSAT core) [6, 7]. Given a correct reference program, a students buggy program, and a counterexample, we translate both programs into SMT formula and assert that the outputs of the reference program and the students program have to be equal on the given counterexample input. Each clause in the formula corresponds to an expression in the program. This formula is indeed unsatisfiable. Given an unsatisfiable SMT formula, we can extract a minimal UNSAT core which is a subset of clauses such that the set is UNSAT, and leaving out any clause in the set will result in SAT. Rosette provides a method to extract a minimal UNSAT core, so we obtain this bug localization method for free using Rosette.

The minimal UNSAT core suggests that we have to change at least one of the clause in the set in order to make the formula SAT. That means we have to alter one of the expressions associated to these clauses to fix the bugs on this particular counterexample. One caveat of using minimal UNSAT core is that if there are more than one mistake in the program, the minimal UNSAT core is likely to only report locations related to one of the mistakes.

4.2 Maximum Satisfiability

Another SAT-based technique is using maximum satisfiability (MAXSAT). Similar to the previous technique, we translate both programs into SMT formula and assert that the outputs of the reference program and the students program have to be equal on the given counterexample input. MAXSAT is used for finding a set of maximum number of clauses that is still SAT. The complement of the set is a set of potential bug locations. To find all potential bug locations, we find the union of the complements of all MAXSAT clauses followed the algorithm in [2]. Unlike the previous technique, this technique should output the correction set that contains potential bug locations for multiple mistakes. We implement the algorithm from [2] in Rosette's backend using our manually written naïve MAXSAT algorithm.

4.3 Results

In this section, we show the results of running the two SAT-based bug localization methods on two programs.

First, Figure 2(a) shows hailstone program with one bug highlighted in red. Figure 2(b) and 2(c) display the outputs from minimal UNSAT core and MAXSAT techniques respectively. Both methods take less than a second to return. In this particular program, the outputs from both minimal UNSAT core and MAXSAT methods contain the actual bug location. However, the MAXSAT method is returns a smaller correction set.

```
def hailstone(n):
    nn = []
    while n != 2:
        nn.append(n)
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    return nn
```

(a) Student program with bugs in red

```
def hailstone(n):
    nn = []
    while n != 2:
        nn.append(n)
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    return nn
```

(b) Bug localization using minimal UNSAT core method

```
def hailstone(n):
    nn = []
    while n != 2:
        nn.append(n)
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    return nn
```

(c) Bug localization using MAXSAT method

Figure 2: Outputs from the two bug localization techniques on hailstone program with one mistake

Next is product program with two bugs, shown in Figure 3(a). Figure 3(b) and 3(c) display the outputs from minimal UNSAT core and MAXSAT techniques respectively. Minimal UNSAT core method takes 6 seconds to return, while MAXSAT method takes 73 seconds. However, we can reduce the MAXSAT method's time by utilizing existing efficient MAXSAT algorithm instead of using our manually written naïve MAXSAT algorithm. Apart from that, minimal UNSAT core technique locates one bug precisely, but does not report anything relevant to another bug. MAXSAT technique also locates the same bug correctly. Although it tries to locate another bug, it fails to do so and instead returns non-buggy expressions that are data- or control-dependent to the actual bug.

The behavior that MAXSAT technique misses actual bugs happens not only when using MAXSAT method but also minimal UNSAT core method. Both methods may only suggest a set of locations such that fixing some will make the program runs correctly on that particular counterexample, but not for all inputs. For example, the output of MAXSAT technique in Figure 3(c) suggests that we can initialize *total* to be the correct output of a given input *n*, and initialize *k* or change the loop condition so that the loop body will not be executed. This will make the program correct on a particular input but not on all inputs.

```
def square(x):
    return x * x

def product(n):
    total, k = 0, 1
    while k <= n:
        total, k = square(k) * total, total + 1
    return total
```

(a) Student program with bugs in red

```
def product(n):
    total, k = 0, 1
    while k <= n:
        total, k = square(k) * total, total+ 1
    return total
```

(b) Bug localization using minimal UNSAT core method

```
def product(n):
    total, k = 0, 1
    while k <= n:
        total, k = square(k) * total, total+ 1
    return total
```

(c) Bug localization using MAXSAT method

Figure 3: Outputs from the two bug localization techniques on product program with two mistakes

In order to make the feedback generation system fix the students program successfully, the bug localization system must output a set of expressions that contain all bug locations, not only some of them in a short amount of time. Therefore, we need to further investigate for a different method that guarantees such property. There are many bug localization techniques the research community has been studies [9]. Statistics-based method is particularly interesting since it is very fast. We can apply this technique and select the first few most likely expressions or statements to be faulty as inputs to our feedback generation system. Another SAT-based technique that might worth considering is angelic debugging [1]. This method may return more precise correction set. However, the algorithm presented in the paper only works with one mistake.

5 Feedback Generation

In this section, we will discuss how we generate feedback to debug the student programs without using a manually written error model.

5.1 Mutations

Our feedback generation system defines a set of heuristic and generic *mutations*. A mutation transforms a concrete expression into a different expression that utilizes *synthesis constructs* to express choices. We extend Rosette to support our three synthesis constructs: `n?`, `v?` and `(either choice1 choice2 ...)` via macros. `n?` represents a symbolic integer constant. `v?` represents a symbolic local variable in the scope. `(either choice1 choice2 ...)` represents a choice from one of the expressions in the series of choices. The synthesis engine is supposed to concretize these symbolic variables to satisfy a given constraint— in this case, the student's and TA's solution should match on all input-output pairs.

In the perspective of feedback generation, each mutation defines how to mutate an expession and fix a bug. For example, one of the mutation that we define is *OffByOne* which transforms an expression `e` to `either(e,e-1,e+1)`. If the bug was due to an off-by-one error, the synthesis engine, using the *OffByOne* mutation, can synthesize the correct expession for this program. We describe all the mutations we define in our feedback generation system as follows.

SameType. $n \Rightarrow n?|v \Rightarrow n?$. Transform a integer constant or a local variable to a symbolic integer or variable respectively.
OffByOne. $e \Rightarrow either(e, e-1, e+1)$. Transform an expression e to either $e-1$ or $e+1$ or keep it as it is.
RangeWithLen. $range(l) \Rightarrow range(either(l, len(l)))$. Students sometimes misuse the function `range` on the list rather than the length of the list. This mutation fixes that particular bug.
SameStruct. This mutation keeps the structure of the expression but recursively mutates the subexpressions and the operators.

$n \Rightarrow n? \mid$
$v \Rightarrow v? \mid$
$e1\ binop\ e2 \Rightarrow SameStruct(e1)\ either(+,-,*,/)\ SameStruct(e2) \mid$
$e1\ comp\ e2 \Rightarrow SameStruct(e1)\ either(<,>,<=,>=,!=,==)\ SameStruct(e2) \mid$
$e1\ boolop\ e2 \Rightarrow SameStruct(e1)\ either(and, or)\ SameStruct(e2) \mid$
$unaryop\ e \Rightarrow either(\ ,!,-)\ SameStruct(e)$

Generic1. $e \Rightarrow either(n?, v?, unyop(n?, v?))$. This mutation transforms any expression to an expression of length 1 (which is a constant integer, local variable, or a unary expression).
Generic2. $e \Rightarrow either(n?, v?)\ either(binop, boolop, comp)\ either(n?, v?)$. This mutation transforms any expression to an expression of length 2 (which can be a binary operation, boolean operation or comparison operation of symbolic variables or constants).
In the future, we plan to add generic mutations to support expressions of arbitrary length.

5.2 Candidate Fix Generation

Figure 4 demonstrates how we generate a set of potential *candidate fixes* for two bug locations and with two mutations. We map a mutation to each bug location, and we try all possible combinations. Each combination generate one *candidate fix*. In this example, we then have 4 candidate fixes.

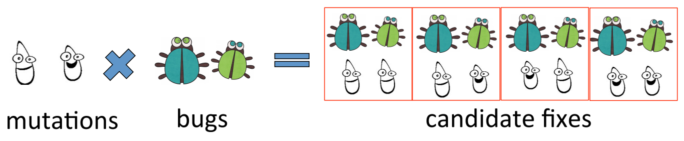
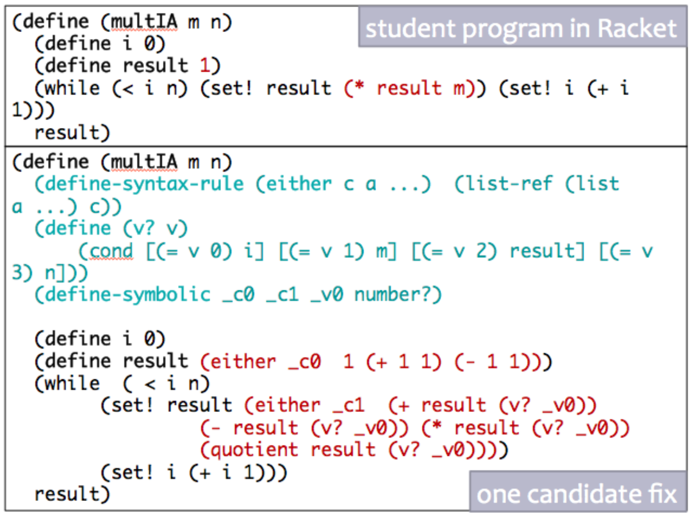


Figure 4: Bug Fixing with Mutations

Figure 5 shows an example of a candidate fix generated from the student program. The bugs in the student program are highlighted in red color. The first bug is that the value of $result$ should be initialized as 0 rather than 1. The second bug is the expression $(* \, result \, m)$ should be $(+ \, result \, m)$, so the correct fix should change the operator from $*$ to $+$.



```
(define (multIA m n)
  (define i 0)
  (define result 1)
  (while (< i n) (set! result (* result m)) (set! i (+ i
1)))
  result)
```

```
(define (multIA m n)
  (define-syntax-rule (either c a ...)  (list-ref (list
a ...) c))
  (define (v? v)
      (cond [(= v 0) i] [(= v 1) m] [(= v 2) result] [(= v
3) n]))
  (define-symbolic _c0 _c1 _v0 number?)

  (define i 0)
  (define result (either _c0  1 (+ 1 1) (- 1 1)))
  (while  ( < i n)
        (set! result (either _c1  (+ result (v? _v0))
                  (- result (v? _v0)) (* result (v? _v0))
                  (quotient result (v? _v0))))
        (set! i (+ i 1)))
  result)
```

Figure 5: An example of candidate fix

The candidate fix is automatically generated using the *OffByOne* mutation on the first bug location and *SameStructure* mutation on the second bug location. The code that is highlighted in blue defines synthesis constructs and symbolic variables used for synthesizing fixes. The first line defines the $either$ construct that chooses the c^{th} element of the series $a....$ The second line defines $v?$ construct that chooses one of the local variables i, m or $result$ according to v. Finally, the third line defines some symbolic variables that are used as parameter in $either$ and $v?$ constructs. The code highlighted in red defines the search space of the program. The first buggy expression 1 is transformed by *OffByOne* to either 1, $(+ \, 1 \, 1)$ or $(- \, 1 \, 1)$. The second expression $(* \, result \, m)$ is transformed to choices of binary expressions where the first argument is $result$, the second argument is a symbolic local variable, and the operator can be either $+$, $-$, $*$ or $/$.

In each candidate fix, our system tries to synthesize for a correct program using $synthesize$ method provided by Rosette, which uses counter-example guided inductive synthesis technique [5] to select the choices of expressions or operations that make the program correct on all inputs.

5.3 Search Strategies

As mentioned in the previous section, given a set of bug locations, we use all of our heuristic and general mutations to generate potential candidate fixes. As not all candidate fixes are actual fixes, we define three search strategies: *parallel*, *mixer* and *priority*, to effectively search for the actual fix.

Parallel. We synthesizes all candidate fixes in parallel. Each synthesizer will then returns whether the synthesis process succeeds, which means the fix is correct, or fails, which means the fix is incorrect. We then return the first fix that is correct.
Mixer. We first combine all the mutations into one general mutation. For example, combining *SameType* and *OffByOne* results in the following code transformation:

$n \Rightarrow either(n?, either(n, n-1, n+1))$
$v \Rightarrow either(v?, either(v, v-1, v+1))$
$e \Rightarrow either(e, e-1, e+1)$

By combining mutation into one, we make the search space for synthesis larger. This search strategy relies on the ability of the solver to prune search space efficiently to arrive at the correct fix.

Priority. In this search strategy, we rank the candidate fixes according to a heuristic and perform the search sequentially. We stop when we find the first correct fix. Specifically, we assign score on the mutations, depending on how likely we think the mutation can correct the bug. We assign score from 1 to 6 to the six mutations *OffByOne*, *SameType*, *SameStruct*, *RangeWithLen*, *Generic1* and *Generic2* respectively. The score of each candidate fix is the sum of scores of the mutations. We then rank the candidate fixes in an increasing order of the scores.

5.4 Results

In this section, we evaluate which search strategy should be adopted. Figure 6 shows the result of running mixer, priority, mixer-priority, and parallel strategies on a 4-core machine. Mixer-priority is running mixer strategy on one thread and priority strategy on another thread, and then we will return whatever the first one that finishes returns. Table 2 shows the properties of all programs we run. The number of candidate fixes run in priority is the number of candidate fixes that are run before we successfully find a correct fix in the priority strategy.

Benchmark	# of candidate fixes	# of candidate fixes run in priority	# of candidate fixes run in priority-reverse	search space	search space ignoring constants
everyOther-s1	4	4	1	2	2
evalPoly-s2	4	1	2	11	11
evalPoly-s4	4	1	2	11	11
hailstone	4	1	2	7	7
product	16	2	5	1080063	171
multA	16	4	5	437	161
computeDeriv	16	1	11	7800117	507
everyOther-s2	64	1	6	4761	225

Table 2: Information on bechmarks. The search space ignoring constants counts a symbolic constant that can take any value as one choice, while the normal search space counts such symbolic constant as many choices (its upperbound subtracted by its lowerbound).

Intuitively, when the number of candidate fixes is not more than number of cores on the machine, parallel strategy should to be best. However, according to the result in Figure 6, the parallel strategy is always worse than the other strategies, even in program 1-4 when the number of candidate fixes is equal to the number of cores. We think that this is because of the overhead of running programs in parallel. As expected, when then number of candidate fixes is high, the time to generate feedback using the parallel strategy is very large because many synthesizers are competing for CPU resources as seen in the last two programs.

Comparing mixer and priority strategy, we observe that the priority strategy returns the feedback before the mixer strategy in 6 out of 8 programs. However, this depends largely on how we assign the priority scores. Figure 7 shows times to generate feedback when using priority strategy with two different ways of assigning the priority scores. The first one is the regular one we described in Section 5.3, and the second one is similar but in a reverse order. We can see that the priority strategy

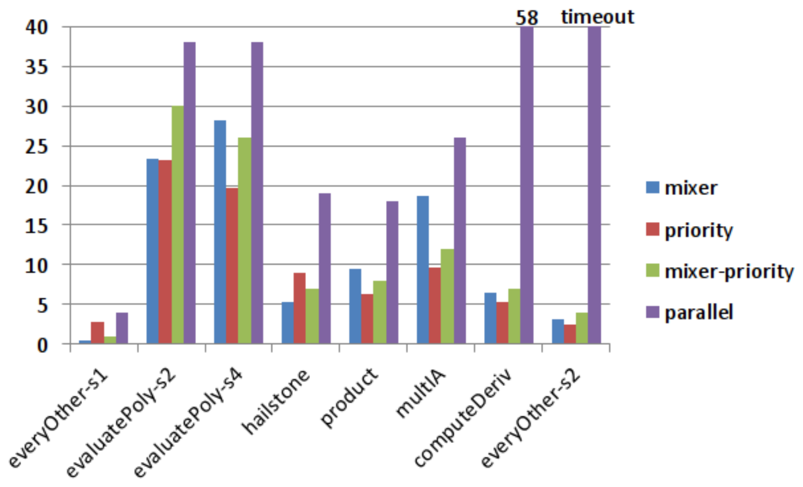


Figure 6: Time to generate feedback using different strategies in seconds.

with the reverse order can be much slower than the regular order and the mixer strategy. Therefore, in practice, we should consider running both the mixer, and the priority strategy in parallel (mixer-priority). We show in Figure 6 that the overhead of running two synthesizers in parallel is acceptable. The mean and the median of time to generate feedback using mixer-priority strategy is 11.9 seconds and 7.5 seconds respectively.

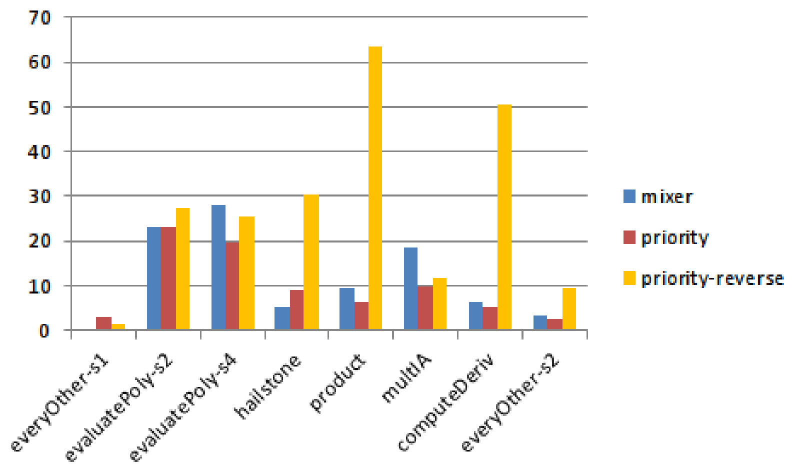


Figure 7: Time to generate feedback using different priority orders in seconds.

Another question we want to investigate is what factor affects the time to synthesize a correct program. Table 2 shows the sizes of search space. The result invalidates the hypothesis that the synthesis time depends on the size of search space. For example, *evaluatePoly-s2*'s size of search space is 11, but the synthesis time is very large. On the other hand, *computeDeriv*'s size of search space is 7,800,117, but the synthesis time is very small. With more analysis, we hypothesize that the synthesis time depends on the complexity of the program. *evaluatePoly-s2* contains one for loop with a symbolic bound. *evaluatePoly-s2* also contains exponent operator which is transformed into a bounded series of multiplications. On the other hand, *computeDeriv* is relatively simpler with only one bounded while loop and no complicated operator. Hence, the time to generated feedback does not seem to correlate with the size of search space but seems to correlate with the complexity of the program.

6 Conclusion

We have presented the first general auto-feedback generation algorithm for introductory Python programs that does not require manually writing an error model. We implemented our algorithm in an efficient and practical tool. The initial evaluation on several programs extracted from the literature of the state-of-art work and CS61A course from UC Berkeley shows that our system is reponsive enough to be useful in a MOOC system. In future, we would like to support full-fledged Python programs, which includes the support for objects and library data structures. We also want to incorporate better bug localization technique to make the system more complete.

References

[1] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 121–130, New York, NY, USA, 2011. ACM.

[2] M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 437–446, New York, NY, USA, 2011. ACM.

[3] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1993.

[4] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI '13*, pages 15–26, New York, NY, USA, 2013. ACM.

[5] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. 2006.

[6] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler. Using unsatisfiable cores to debug multiple design errors. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, GLSVLSI '08, pages 77–82, New York, NY, USA, 2008. ACM.

[7] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming &#38; Software*, Onward! '13, pages 135–152, New York, NY, USA, 2013. ACM.

[8] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI '14*, 2014.

[9] W. E. Wong and V. Debroy. A survey of software fault localization, 2009.

Computational Tools for Music Theory Pedagogy

Chris Shaver

May 16, 2014

Abstract

This report introduces *Counterdot*, a Python library for the representation and manipulation of musical structures aimed at the Music Theory pedagogy. The basis of this library is an ontology of musical objects defined by a collection of constant object classes. An embedded expression language is defined over these classes to allow composite objects to be constructed rapidly with a simple syntax. This report shows how to use this language to generate random chords, chord progressions, and modulations from a flexible set of specifications. Applications such as these can be used to generate examples and exercises of pedagogical value in music theory classes, and ultimately develop tools for Massive Open Online Courses (MOOCs).

1 Introduction

In a class on Music Theory or Harmony, a student must learn how to define, recognize, and construct structures in music composition. Developing these skills involves formal exercises with these structures. A discussion of this pedagogical process is given at length in Schoenberg's *Theory of Harmony* [5]. The initial aim in this learning process is for the student to become fluent in a formal language used to describe and discuss composition in a sophisticated manner. Subsequently, a study of music composition is built off of this language. It is not enough, therefore, to simply understand how to interpret the language. To the contrary, it is greatly advantageous to recognize patterns quickly and construct musical structures intuitively.

In order to become proficient with the basic structures of music theory, examples and exercises are necessary. Devising these examples and exercises by hand is tedious, prone to error, and may also involve the unconscious introduction of undesired systemic patterns that detract from learning the appropriate ones. This difficulty is even greater in the context of Massive Open Online Courses (MOOCs) for which it is of great benefit to be able to generate novel exercises that can be automatically evaluated. The automatic generation of exercises is also particularly useful for students who wish to drill themselves on problems in order to strengthen intuition and comprehension.

In this report, the Python library *Counterdot* will be introduced as a platform for the development of applications for music theory pedagogy. This library features, at its core, an ontology and expression language, both of which will be discussed. It will then be shown how *Counterdot* is used to generate random musical structures for examples and exercises. Specifically, random chord voicing, the construction of chord progressions, and modulation generation will be described as applications.

Online versions of the examples shown here can be found at:
`http://www.eecs.berkeley.edu/~shaver/counterdot/examples/web/`

2 Background

Like most programming language, in music theory and composition, there is a gap between syntax and semantics. The same musical pitch, for instance, arguably the semantics of a musical note, can be described different ways. An $A\flat 4$ and a $G\#4$ both denote the same pitch. In music terms, they are *enharmonic*. They also indicate to a performer, on any common instrument, the actuation of the same sound. It is tempting therefore to ontologically label this pitch with a single notation, and to generally make the syntax of a language for music representation bijectively mapped into its sonic realization. Doing this is adequate and efficient when simply storing the actuation of pitches as data, as is done in MIDI.

However, in music theory, two such notes maintain different names because they not only denote pitch, but more, connote things about their compositional context: what key they are in, what chords they belong to, etc... This syntactic surplus is present on many more levels than that of the individual notes. Two different chords, for instance, may be composed from the same notes (let alone pitch classes). An *A Minor 7* chord and a *C Major 6* chord both contain A, C, E, and G. Moreover, if tones are omitted from a chord, this overlap becomes even more pronounced.

For this reason it is beneficial to maintain the full resolution of musical syntax when representing musical objects in the context of music theory. It is also beneficial to provide a route for the unambiguous construction of these objects, so that none of these conflations are introduced. A *chord* is constructed, unambiguously, from a root note and a set of intervals. In contrast, it is not constructed, but inferred from a set of notes, since these notes may not define it uniquely.

The *Counterdot* library provides a clear syntactically unambiguous representation for musical objects that maintains all syntactic differentiations. It also provides for each composite object, a unique mechanism of construction from simpler objects that can be decomposed uniquely.

Along the above lines, some important comparisons can be drawn to other systems of musical representation. The *Music21* Python library [3] also provides a rich ontology for musical objects. However, *Music21* is focused towards musicology and analysis, and is consequently more descriptive than constructive. It does not provide a clear, unambiguous syntax for the construction of musical objects and makes implicit inferences. A **Chord** in *Music21*, for instance, can be constructed from a simple set of notes. The object then has many methods that infer its properties. It cannot be decomposed in a clear way into unambiguously simpler components.

Another example of a sophisticated music representation system is that of the *Antescofo* system [2], which focuses on score following and live improvisation. Although one of the aims of this system is pedagogy, its pedagogical focus is more towards improvisation and performance. It is also its own system, rather than a library, and thus it is hard to integrate it into a larger system used in a MOOC or online course software. *Counterdot*, in contrast, being a Python library, can easily be used in the development of web applications and other interfaces. *Musique Lab 2* is more suited to music theory pedagogy [4], but is also a monolithic system that cannot easily attached to web applications or developed into simple scripts that might, for instance, generate printed exams or homework assignments.

At this stage in development, *Counterdot* does not have any utilities to export structures into notational formats. For this reason, in the development of *Counterdot*, *Music21* is used to export *MusicXML*, and *VexFlow 2* [1] is used to notate structures in *HTML5*.

3 The Counterdot Language

The *Counterdot* is a library, written in Python, consisting of a language of classes for the representation of musical objects, a library of common constants, a mathematical embedded expression language for constructing and combining these objects, and algorithmic tools for the generation of random objects, examples, and exercises. The first three components will be discussed as follows, whereas the algorithmic tools will be discussed in the subsequent section on applications.

3.1 Ontology

The Ontology of *Counterdot* consists of a collection of constant (immutable) object classes. A group of primitive classes form the basis from which the other classes are constructed, largely as products of these classes. More than just encapsulating the representations of musical objects, these constant classes form a system of types that can be used verify the correct construction of expressions involving their corresponding objects. Even if the constituting information is the same for two classes, their difference is used to distinguish between them ontologically. For instance, an **Interval** and a **RelVoice** both consist of an **Index** and an **Octave**. Nevertheless, these classes are made distinguishable because they serve different roles and should be used in different contexts.

Although constructors of these classes are given default parameters, there is no implicit coercion between a component of an object of a particular class and an object of this same class. For instance, an object *v* of class **Voice** consists of a **Note**, *n*, along with an **Octave**, *m*. And although one can construct a **Voice** from the note, **Voice**(n), with a default **Octave** of 0, the **Note** will never be conflated with the **Voice**.

The identity of each object, within its class, is established by the identity of its components. In principle, this could be achieved through an inductive definition of equality, though here it is more expediently implemented through the memoization of objects. A secondary benefit of this memoization is that many classes have a collection of prevalent instances predefined as constants. Consequently, secondary features of these instances, such as the component **Voices** of a **Voicing** only need to be calculated (or declared) once, and are often already determined with the instantiation of the collection of provided constants. Since the objects are all immutable, memoization limits the instantiation of objects to only those that have not yet been used. In the terms of Python, this means for objects *a* and *b*, $a == b \Leftrightarrow a$ **is** *b*.

The ontological types of elements in Counterdot can be split into a group of Scalar Types and a group of Composite Types, each of which will be discussed in the following.

3.1.1 Scalar Types

The primitive scalar types consist of the type **ScaleCard**, representing the cardinality (number of components) of structures like Scales and Chords, along with a collection of indexing types. The most basic indexing types are **Diatones**, **Diadexes**, and **Degdexes**. **Diatones** index the base notes of the diatonic scale, C, D, E, F, G, A, B, without accidentals. These correspond to the notes in the C Major scale, and the names of the white keys on a piano. This collection is used as an absolute origin against which all other structures are defined. **Diadexes** are relative indices of the diatonic scale, representing the number of steps from one element to another cyclically. There are consequently 7 of these elements. **Degdexes** index the elements of musical structures of a particular cardinality. A **Degdex** object is thus is composed of a **ScaleCard** object and an integer, the identity of which is treated as modulo the cardinality. The notation used for these **Degdex** objects is n/M, where n is the index and M is the cardinality. For objects of some cardinality M (M distinct elements), such as an **M** note scale, **Degdex** objects n/M are used to reference the **n**th element.

Two additional primitive scalar types are used to modify the index types. The first of these is the **Shift** type, which represents accidental shifts up or down numbers of half steps. Combining the above three indexing types with a **Shift** objects results in objects of the type **Note**, **Index**, and **Degree**, respectively. The second modifier is the **Octave** type, used to place a **Note**, for instance, in a particular octave. Combining the index types, and those with **Shifts**, with an **Octave** results in objects of the type **Voice**, **RelVoice**, and **Placement**, respectively. Objects of type **Voice** have enough information to characterize the notational position of a musical note.

The type **Placement** is worth explaining directly as well. An object of this type, notated $n/M(A)@k$, has four components: a cardinality *M*, an index *n*, a shift *A*, and an octave *k*. An object of this type, when applied to a scale of cardinality *M*, can extract the *n*th element, shift it with the accidental *A*, and place it in the *k*th octave. Concretely, suppose a position $3/7(1)@3$ is applied to the F-Major scale. The resulting **Voice** is an $A\#3$. It should be noted that the octave here is relative to the root of the scale, hence $6/7(0)@3$ applied to the F-Major scale would be *E4*, not *E3*.

Finally, the type **Interval** is, like a **RelVoice**, constructed from an **Index** and an **Octave**, and also used to represent the harmonic difference between two **Voices**. This is a separate type from **RelVoice**, because intervals have a special nomenclature, specialized operations, and is ontologically used in different contexts. In the harmonic structure of Chords, although a *Major 2nd* and a *Major 9th* away from some note, for instance a *C*, refer to the same note, in this case a *D*, there is a difference in how the 9th is conventionally voiced.

3.1.2 Composite Types

The composite types in *Counterdot* consist of musical objects built from collections of scalar types:

ScaleTypes are lists of **Index** elements representing types of scales. Each **Index** element in the list indexes the diatonic major scale with an additional shift. Some examples of **ScaleTypes** are as follows:

$$\begin{aligned} \textit{Major} &= 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7 \\ \textit{Minor} &= 1,\ 2,\ 3\flat,\ 4,\ 5,\ 6\flat,\ 7\flat \\ \textit{Pentatonic Minor} &= 1,\ 3\flat,\ 4,\ 5,\ 7\flat \\ \textit{Diminished Triad} &= 1,\ 3\flat,\ 5\flat \end{aligned}$$

ChordTypes are sets of **Interval** elements representing types of Chords. Each interval is relative to the root of the Chord. A *Major Triad* is therefore defined $[M3,\ P5]$, whereas a *Dominant 9th* Chord is defined $[M3,\ P5,\ m7,\ M9]$. If a root **Note** is conjoined with a **ScaleType** or **ChordType** the result is, respectively, a **Scale** or **Chord** object. Straightforwardly, combining the *Dominant 7th* **ChordType** with an *A#* **Note** results in an *A# Dominant 7th* **Chord**.

Both a **ChordGroup** and a **Layout** are constructed from lists of **Placements** of equivalent cardinality. However, these two types have different functions. Suppose the **Placements** in a **ChordGroup** object are denoted $n_i/M(A_i)@k_i$. Given a scale S with cardinality M, this **ChordGroup**, applied to scale S will produce a **ChordType** object consisting of intervals

$$S[n_i/M(A_i)]@k_i - S[n_0/M(A_0)]@k_0$$

An example of this would be the **ChordGroup** $(1/7@0,\ 3/7@0,\ 5/7@0)$. If this is applied to a Major scale, one gets a major triad, whereas if it is applied to a Minor scale, one gets a minor triad. The **Octave** component is necessary here to express the group of 9th chords, for instance, where the 2nd element of the scale is included in the second octave. This mechanism is particularly useful as a means to extract the harmonized chords of a scale. This will be elaborated on further in a later section.

A **Layout**, in contrast, uses the constituting **Placements** to map the elements of a **Scale** or **Chord** into particular octaves. The result of this mapping is the construction of a **Voicing** of this **Chord**. In short, a **Voicing** is a **Layout** applied to a **Chord**. For instance, consider a **Layout** with the following **Placements**

$$(2/4@0,\ 1/4@1,\ 3/4@1,\ 4/4@1,\ 3/4@2)$$

This sequence of **Placements** denotes a voicing of a 4 note chord (such as a 7th chord) in first inversion with a repeated third element (the 5th in a 7th chord) in an upper octave. Applied to a *G Minor 7th* **Chord**, one gets the **Voicing** constituted of the following **Voices**

$$(B\flat@0,\ G@1,\ D@2,\ F@2,\ D@3)$$

Note that the octaves of the notes are adjusted to account for the offset of the root note (here, G). If instead, the same **Layout** is applied to a $D\flat$ *Dominant 7th*, the resulting **Voicing** is

$$(F@0,\ D\flat@1,\ A\flat@1,\ C\flat@2,\ A\flat@2)$$

This separation between the **Chord** and the **Layout** plays a useful role later in generating random chords, since the two components can be generated separately, then combined. Indeed, a **Voicing** is constructed out of these two objects. When it is instantiated, a corresponding list of **Voices** is produced. This construction is unambiguous from a harmonic perspective. In contrast, from a simple collection of **Voices** it is not necessarily clear which note is the root and thus it is not necessarily clear which type of Chord is being Voiced. If the Voicing omits components of the Chord in its Voicing, it is even more ambiguous. Going from the **Voices** to the **Voicing** is therefore a matter of inference.

Another variation on **ChordType** is the type **HFunction**, which represents a *Harmonic Function*. Objects of this type are constructed from a **ChordType** and a **Degree**. When applied to a **Scale** the result is a **Chord** of the given **ChordType** with the given **Degree** of the **Scale** as its root. For example, an **HFunction** denoting the $3/7\,\mathbf{Minor}$ chord applied to the *D Major* scale would result in an *F# Minor* chord.

3.2 Embedded Expression Language

Given the capability of Python to overload mathematical operations, and other operations such as array indexing, when one of the operands is of a defined class, the class of *Counterdot* have several overloaded operations defined on them, constituting an Embedded Expression Language that can be used to construct and manipulate musical objects.

For scalar types, the operation of addition is used to denote the combination or accumulation of components. The objects of types **Diatone**, **Note**, and **Voice** are *absolute* scalars, since they refer to specific musical objects, while objects of types **Shift**, **Octave**, **Diadex**, **Index**, **RelVoice** are *relative* scalars, since they refer to differences or offsets in musical objects. In the expression language, *relative* objects can all be added, either to each other or to *absolute* objects. However, *absolute* objects cannot be added to each other, since the meaning of this would be unclear. The support for addition is summarized in the table [tab:addition], which also shows the type of the result when given heterogeneous operands.

Some examples of these operations are as follows:

$$\begin{aligned} A + \flat + 3\flat &= C\flat \\ 3 + 3 &= 5\# \\ G\# + @3 + 6\flat &= E@4 \\ \# + @2 &= 1\#@2 \end{aligned}$$

Other operations, such as subtraction, are defined for scalar types. Composite types have addition defined to denote transposition. For many of the types, comparison are defined. In contrast with some other musical representation systems, the direct form of these comparisons are syntactic rather than pitch-oriented, hence $A\# < G\flat\flat$, even though the pitches are ordered oppositely. This order is more useful when handling formal harmonic properties of these objects.

4 Pedagogical Applications

A key to both generating novel examples and exercises is being able to generate random representatives from different sets of musical objects. The *Counterdot* language makes this easy because objects are either simple, primitive ones, or constructed inductively from other objects. In the primitive case, a random object can be chosen uniformly over the set, if it is finite, uniformly over some finite subset, or chosen by some other distribution. Other composite objects can be generated randomly by inductively choosing at random their components. Since the components of objects are independent, randomly choosing their constituents uniformly gives a uniform random choice over the objects. Nevertheless, more complicated constraints could be implemented in principle.

4.1 Random Chord Generation

A particularly useful example of the random generation of musical objects is the random generation of chord voicings. Chord identification exercises involve the presentation of chord voicings notated on a musical stave as a set of voices (notes in particular octaves). The goal of chord identification is to correctly infer from the set of voices the root note and the type of chord. Using a random chord generator, exercises like this could be automatically produced and easily automatically checked.

Using the ontology of *Counterdot*, generating a random chord voicing involves three parameters: a set $\mathcal{T}$ of **ChordTypes**, a set $\mathcal{N}$ of **Notes**, and a layout algorithm **L**. The first two of these can be given outright as finite sets, or generated in many ways. By randomly choosing an element $t \in \mathcal{T}$ and an element $n \in \mathcal{T}$, a random **Chord** $c = \mathbf{Chord}(\mathtt{t},\, \mathbf{n})$ can be constructed from these two data. The third parameter **L** is more involved and may itself involve parameters. Generally, **L** is a function from **Chords** to **Layouts**. Applied to the randomly chosen **Chord**, c, a **Layout**, $L(c)$, can then be combined with the chord to form a **Voicing**, $\mathbf{Voicing}(c,\, L(c))$.

There are many ways to construct a layout algorithm. One particular way will be described here, which is implemented as part of the *Counterdot* library. This method only depends on the cardinality of the given **Chord**. It takes additionally three parameters: a maximum range k in octaves, a number of repeated notes from the chord W, and the inversion q. The last parameter, inversion, denotes the element of the chord to appear in the lowest position in the voicing. For instance, if $q = 0$, a *G Dominant 7th* **Chord** with be voiced with *G* as the lowest note (this incidentally referred to as *root position*). If $q = 2$, on the other hand, *D* will be the lowest note.

Listing 1: Random Layout Generation 1

```
def generateLayoutAlgorithm1(k, W, q, chord):
  bdeg = chord.card.degdex(q)
  octrange = lambda d: (lambda i: range(i, k+i))(int(d < bdeg))
  potset   = lambda d: {d + Octave(x) for x in octrange(d)}
  potentials = {d: potset(d) for d in chord.card.degdices}
  essentials = set(potentials.keys())

  bass = bdeg + Octave(0)
  essentials.remove(bdeg)
  potentials[bdeg].remove(bass)

  placements = [bass] + [drawFrom(potentials[e]) for e in essentials]

  remanders = reduce(lambda a, x: a | x, potentials.values())
  placements += drawFrom(remanders, W)
  return Layout(sorted(placements))
```

The algorithm code is given in listing 1. It first generates a set of possible placements for each component of the chord in the range starting from lowest note of the chord, determined by the inversion q, up to (and excluding) the same note k octaves higher. The lowest note is then added to the layout and removed from its corresponding set of possible placements. For each of the other sets, representing the other components in the chord, a placement is chosen at random, added to the layout, then removed from its corresponding set. This is necessary to ensure that the chord has at least one instance of each of its components. The sets containing the remaining placements are then combined into a single set of possible repetitions, and W elements are drawn randomly from this set.

Using this method a wide variety of chord voicings can be generated that are valid chords by construction. In Figure 1, some examples of random chord generation are shown using this first random layout algorithm. The common parameters are as follows

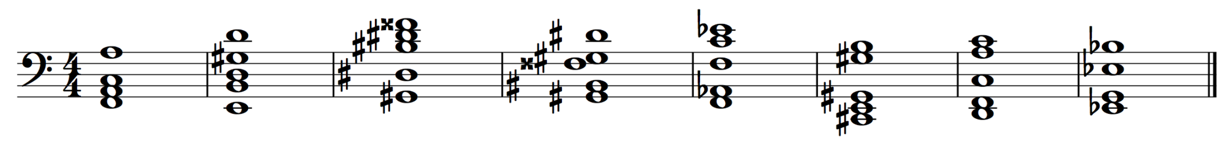
$$\mathcal{N} = \{C, C\#, D\flat, D, E\flat, E, F, F\#, G\#, A\flat, A, B\flat, B\}$$
$$\mathcal{T} = \{\mathbf{M}, \mathbf{m}, \mathbf{M7}, \mathbf{m7}, \mathbf{D7}\}$$

The remaining parameters for the layout are shown with their corresponding figure. To add additional variation to the generation, the parameters can be randomly selected as is shown in examples 1c and 1d. Here the range is fixed, but the inversion and the number of repeats are chosen randomly from given sets.

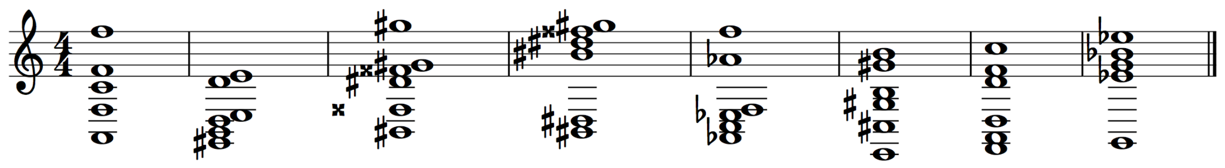
4.2 Chord Progressions

Another kind of exercise relevant in music theory is one in which a sequence of chord voicings is given. This sequence forms a *Chord Progression*. In addition to identifying the chords, a student must determine what *Harmonic Function* is played by each chord in the *Key*. These Functions are usually denoted by roman numerals with additional conventional annotations and modifications, and the sequence of them denotes a particular chord progression. A common example of such a progression is

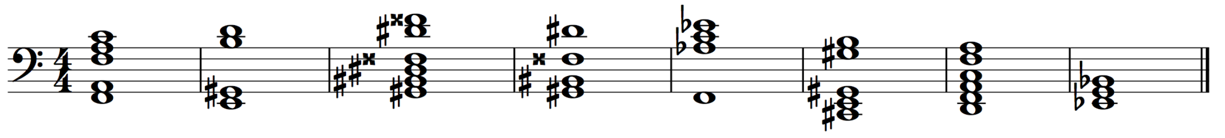
$$\mathbf{I}, \mathbf{vi}, \mathbf{IV}, \mathbf{V}, \mathbf{I}$$



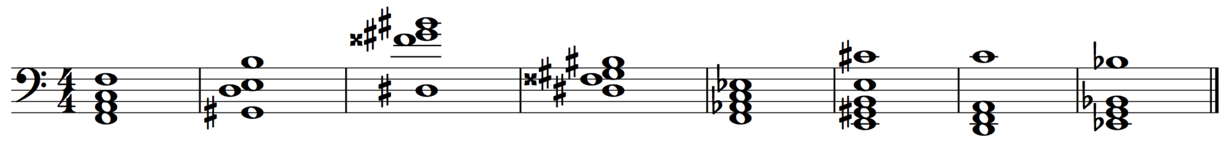
(a) $k = 2, W = 1, Q = 0$



(b) $k = 3, W = 2, Q = 1$



(c) $k = 2, W = \{0,\ 2\}, Q = 0$



(d) $k = 2, W = \{0,\ 1\}, Q = \{0,\ 1,\ 2\}$

Figure 1: Random chord generation with different parameters for layouts. In all examples, the chords voiced are: F Major, E Dominant 7th, $G\#$ Major 7th, $G\#$ Major 7th, F Minor 7th, $C\#$ Minor 7th, D Minor 7th, $E\flat$ Major

Listing 2: Construction of chord progression

```
Prog1 = lang.Degdices.words("I I V I VI IV V I")
chordgroup = lang.ChordFamilies.seventh
harmscale = lang.ScaleTypes.Major
scale = lang.Scale(harmscale, lang.Notes.D)
HFuns = [harmscale.hfun(chordgroup, d) for d in Prog1]
chords = [h(scale) for h in HFuns]
```

The value of the roman numeral in the function refers to a chord rooted at the corresponding element of a scale determined by the key. The type of chord is usually one that has its components inside of the key. For instance, in the key of *G Major*, corresponding to the *G Major* scale, the **V** chord is a *D Major* chord, since *D* is the 5th note in the *G Major* scale and the notes *F#* and *A* are both in this scale.

In the *Counterdot* language, a chord progression can be represented directly by a sequence of **HFunctions**. This sequence can then be applied to a particular scale, resulting in a sequence of **Chords**. Another useful way to represent a progression is to simply give a sequence of **Degdexes**, referring to the degree upon which to build each chord. The **ScaleType** class has a method **hfun** that determines the **ChordType** in a **ChordGroup** rooted at a given **Degdex** of the **ScaleType**. Take as an example the sequence of **Degdexes**

$$1/7, 6/7, 4/7, 5/7, 1/7$$

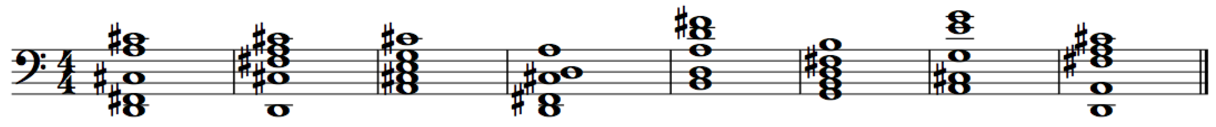
Mapping over this sequence with a *Major* **ScaleType** and the **ChordGroup** of *Seventh* Chords, one gets the following sequence of **HFunctions**

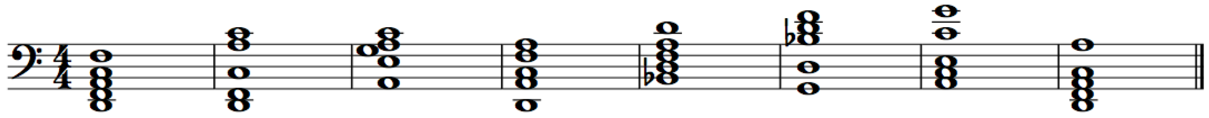
$$\mathbf{I} - Maj7,\ \mathbf{VI} - min7,\ \mathbf{IV} - Maj7,\ \mathbf{V} - Dom7,\ \mathbf{I} - Maj7$$

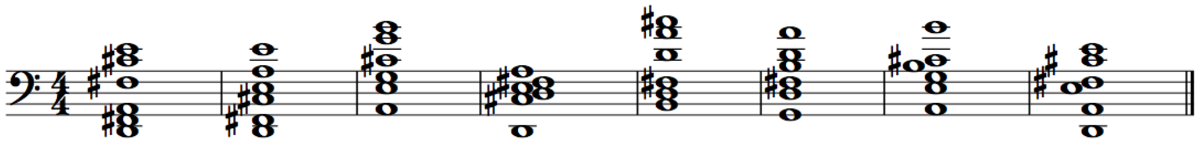
In other words, these are chords harmonized with the *Major* **ScaleType**. The **HFunctions**, either given explicitly or generated from the **Degdexes** of a **ScaleType**, can then be applied to a **Scale**, for instance the *G Major* scale, resulting in a sequence of chords: $GM7$, $Em7$, $CM7$, $D7$, $GM7$. These chords can then be given **Layouts** with a layout algorithm in order to produce **Voicings**. This layout algorithm could be, for instance, the one described above.

By randomly selecting **Scales**, randomly selecting from a set of predefined progressions, and randomly generating layouts, a variety of progression identification exercise can be generated. If more variation is desired, the progression itself can be randomly generated by randomly selecting a sequence of **Degdexes**. As shown above, this sequence can be transformed into the appropriate harmonized chords for the **ScaleType**. Randomly generating **HFunctions** themselves would result in unrealistic examples that do not conform to the key center of a piece.

An example of this construction of harmonized chord progressions from **Degdexes** is accomplished in *Counterdot* with the code given in listing 2. Once the chords are generated, voicings can be constructed, potentially using the same methods described above. In Figure 2, this method is used to generate voicings of chord progressions. In examples 2a, 2b, 2c, the same fixed progression, (**I**, **I**, **V**, **I**, **VI**, **IV**, **V**, **I**), is used to generate the sequence of chord voicings. In each case the **Chords** produced belong to the **ChordGroup**, $\mathcal{G}$, and are chosen to harmonized the corresponding elements of the **Scale**, S. In 2a, the **Scale** is *D Major* and the **ChordGroup** is the *Seventh* chords. In 2b, the **Scale** is changed to *D Minor*. Consequently, the types of chords differ corresponding to the differences in notes between the two scales. The example 2c differs from the first in that the chords generated are *Ninth* chords rather than *Seventh*. In the last example 2d, **Degdexes** forming the progression are chosen randomly, except for the final one which is the I.


(a) S = *D Major*, $\mathcal{G}$ = *Seventh Chords*


(b) S = *D Minor*, $\mathcal{G}$ = *Seventh Chords*


(c) S = *D Major*, $\mathcal{G}$ = *Ninth Chords*

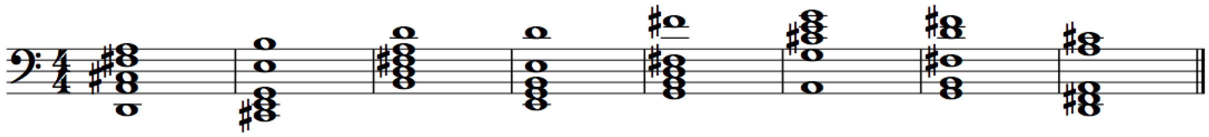

(d) S = *D Major*, $\mathcal{G}$ = *Seventh Chords*, with a random progression
II, **II**, **III**, **VI**, **VII**, **V**, **VII**, **I**

Figure 2: Voiced chord progressions in **Scale** S, using members of **ChordGroup** $\mathcal{G}$

Listing 3: Automata-based Modulation

```
degs = lang.Degdices.words
transmap = map(degs, [
    "II", "II IV VI", "I II III IV V VI VII", "I III VI", "III"
])

init, inc = lang.Indices.words("i2b i5")
states = reduce(lambda a, x: a + [a[-1] + inc], range(11), [init])
language = product(states, states, degs("I II III IV V VI VII"))
exstates = [states[0], states[0]] + states + [states[-1], states[-1]]
windows  = zip(*[exstates[n:] for n in range(0, 5)])
transit  = [(n, t) for n in range(0, 5) for t in transmap[n]]
transn   = lambda m: {(m[2], m[n], t): m[n] for n, t in transit}
transfun = {m[2]: transn(m) for m in windows}

start, term = lang.Indices.i1, [lang.Indices.i1]
chordProgAuto = DFA.DFA(states, language, transfun, start, term)
```

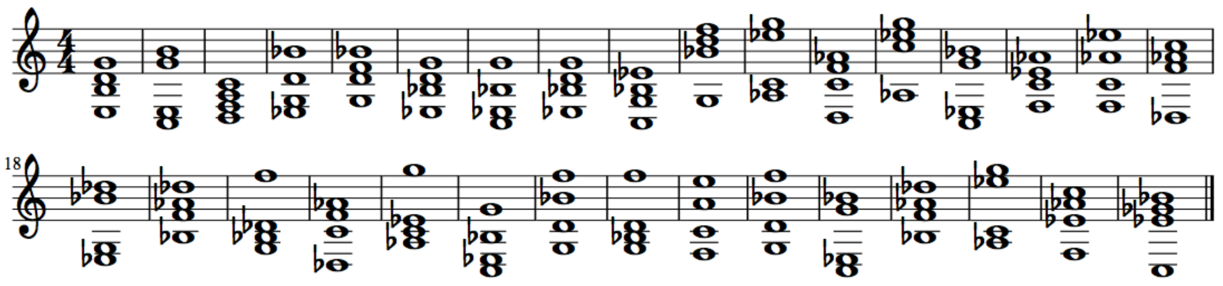
4.2.1 Automata-Based Modulation

More complex exercises involving chord progressions would involve more complex constraints on the sequences of chords. A particular example of this are progressions that involve *Modulation*, which is a change of key center. Realistic instances of modulations in music involve the use of chords that are harmonized to multiple keys to pivot between them.

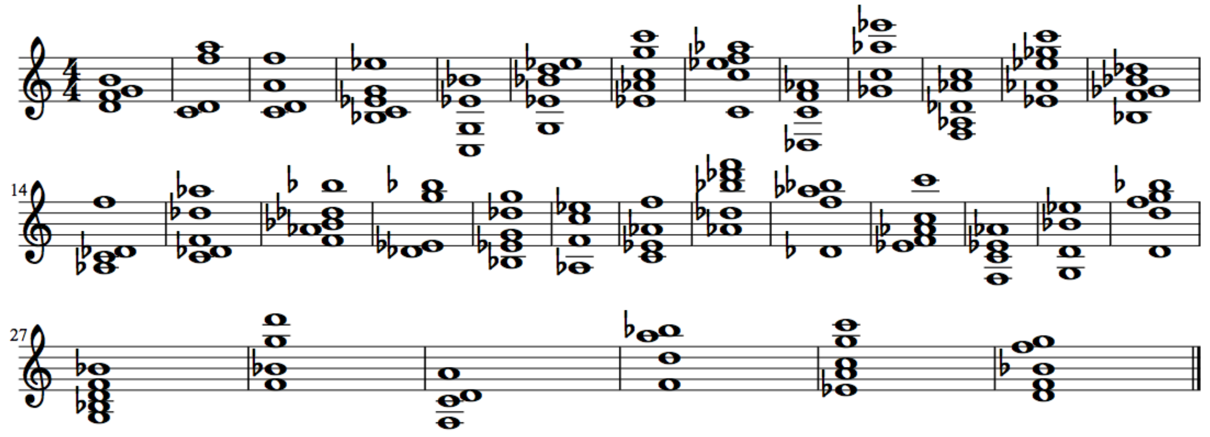
In order to generate (or evaluate) modulating chord progressions with *Counterdot*, a Finite State Automata (FSM) can be used. Such a FSM has key centers as its states and harmonic functions (chords) as its transitions. A self-transition, specifically, corresponds to a chord that does not pivot into a new key center, while a transition between key centers must involve a chord that pivots between them. Such a chord has all of its components in both keys. An example of such a chord is the *C Major 7* **Chord**, which is both the *I* Chord in the *C Major* **Scale** and the *IV* Chord in the *G Major* **Scale**.

As it is stated, this automata is, of course, non-deterministic, since a pivot chord can be both used to pivot to another key center or simply a self-transition remaining in the same key. A simple random walk through the automata would not therefore necessarily give a uniform distribution of words. However, for the context of music pedagogy, the conjunction of chord choice and modulation target would make the transitions deterministic, and might be a more valuable anyways. A composer presumably chooses to modulate when choosing a chord to construct a harmonic progression, rather than drawing intuitively from a collection of acceptable progressions blind to the modulations. Moreover, these modulations, in practice, show up in other ways than just chord choice such as non-chord tones and melodic lines.

In listing 3, an algorithm is given for constructing the automata. In this case, the transition structure is set up for *Seventh* chords as pivots. The labels for the transition include the current key, the subsequent key, and the **Degdex** upon which to construct the chord. Two examples are given in Figure 3. Using the automata, and doing random walks though it, the rest of the procedure for producing these examples is the same as in the above examples for chord progression generation.



(a) Automata modulation example



(b) Automata modulation with random inversions

Figure 3: Generated modulating chord progressions using an automata.

5 Conclusions

As can be seen from the discussed applications, the *Counterdot* library provides a simple useful faculty for music pedagogy, particularly in the case of MOOCs where the demand for automatic content generation is inevitable. A simple, proof-of-concept web application has already been created for chord recognition exercises using a *Twisted* Python server and the *Counterdot* library as a back-end. The front end is written in HTML5, Javascript, and CSS with the aid of the *VexFlow 2* library [1]. However, these same tools can be used in a much wider context to experiment with generative music and algebraic analyses of harmony.

Thus far, the basic components expounded here have all been implemented. Nevertheless, the ontology will require further scrutiny to fine-tune its basic relationships, and the implementation sill has some clear features to complete, particularly in the embedded expression language. Further work will be done to make the library more complete and stable, and as well develop more algorithms for generation and inference. Additionally, work can be done to develop the web application into a tool ready for practical use in the context of an actual class or MOOC.

References

[1] Mohit Muthanna Cheppudira. Vexflow 2. `https://github.com/0xfe/vexflow`, 2010.

[2] Arshia Cont. Antescofo: Anticipatory synchronization and control of interactive parameters in computer music. In *International Computer Music Conference (ICMC)*, 2008.

[3] Michael Scott Cuthbert and Christopher Ariza. Music21: A toolkit for computer-aided musicology and symbolic music data. In *ISMIR*, pages 637–642, 2010.

[4] Vincent Puig, Fabrice Guédy, Michel Fingerhut, Fabienne Serrière, Jean Bresson, and Olivier Zeller. Musique lab 2: A three level approach for music education at school. In *Proc. of the International Computer Music Conference*, 2005.

[5] Arnold Schoenberg. *Theory of harmony*. Univ of California Press, 1978.

Problem Generation for DFA Construction

Alexander Weinert

May 16, 2014

Abstract

Teaching the construction of DFA to computer science students relies in great part on practice problems, in which the student is asked to construct an automaton for some given language. Nowadays, these practice problems are constructed by humans in order to teach certain general concepts that, once understood, can be reused in the construction of other DFA. A problem arises if the student finishes all their practice problems, but still has not understood the underlying concept. We present a method to generate practice problems using one concrete example problem as the input. During the construction we make sure that the resulting problem has the same level of difficulty and exercises the same concepts as the original problem. We also present an evaluation of our algorithm on 20 examples from a well known textbook on automata theory.

1 Introduction and Motivation

Nearly every student of computer science is taught the concept of Deterministic Finite Automata, or DFA for short, at some point during their studies. The teaching of this subject usually consists of a formal definition, some examples of automata that recognize simple string languages, and some exercises for the students, which are often of the form "Construct a DFA $\mathcal{A}$ that recognizes the regular language L". L is most often defined in English.

It is then usually assumed that the student understands the techniques that are used when constructing automata. However, this may not be the case. It may be that the student would like to have more practice problems that exercise the same principles. In this case the student usually has to rely on problems from previous semesters or from other courses for more exercise. This poses a set of new difficulties, since the older problems may use different notations or teach concepts in a different order, which would further complicate the learning process.

We present a technique that takes a regular language L as input and outputs a set of regular languages in such a way that the minimal automata that recognize the output languages use the same concepts as the minimal automaton that recognizes L and are of similar complexity.

Solving problems of this form, i.e., constructing the automaton for a given language L is very simple if the language L is defined as a formula in Monadic Second Order Logic, or MSOL for short. For every MSOL formula we can construct an automaton that recognizes exactly the language that is defined by the formula, and vice versa [Tho97]. Grading student solutions for these problems has been investigated in [ADG$^+$13], which forms the basis for most of our work in this project.

In [SGR12], the authors research the automated generation of algebra problems from a given problem. A more general treatment of problem generation can be found in [SSG12], where the topic is investigated for a massively open online course in embedded systems. Problem generation has also been investigated in [AGK13] in the context of natural deduction. Generating feedback for faulty student solutions has been not been researched for DFA construction exercises, but there exists an approach for introductory programming assignments in [SGSL13].

2 Problem Definition

We only concern ourselves with tasks of the form “Construct a DFA that recognizes the language L”, where L is some regular language. Our goal is to construct a set of tasks of the same form, such that the student exercises the same principles for the construction of automata when solving the new problems. Since we only concern ourselves with DFA, we are going to use the terms “automaton” and DFA interchangeably for the rest of this report.

Since the only possibility for variations in this kind of tasks is the choice of the regular language L, the task reduces to the following: Given some regular language L, construct a new regular language L', so that the automata that recognize L and L' use the same concepts.

In order to make the approach feasible, we only consider the minimal automata that recognize L and L'. We also assume that the language L is given as a formula in MOSEL. This logic provides syntactic sugar over the well known Monadic Second Order Logic, which is in turn equivalent to finite automata. Thus, MOSEL formulas correspond to regular automata as well. The full definition of MOSEL as well as the transformation between automata and MOSEL formulas is detailed in [ADG+13].

Thus, our final problem is as follows: Given some MOSEL formula φ, construct another MOSEL formula φ', such that the corresponding minimal automata $\mathcal{A}_\varphi$ and $\mathcal{A}_{\varphi'}$ use the same concepts for recognizing their respective language and are of similar complexity.

3 Approach

Our approach works in three steps: Abstraction, Concretization and Filtering. In the first step, we *abstract* the given MOSEL formula into a CHOICE-MOSEL formula, which represents a set of MOSEL formulas, including the original one. We then *concretize* the CHOICE-MOSEL formula, i.e., we construct the set of all MOSEL formulas that are represented by the CHOICE-MOSEL formula. In a final step we *filter* the resulting MOSEL formulas in order to remove formulas whose minimal automaton differs too much from the minimal automaton of the original formula.

Since we are exclusively dealing with the parse tree of a formula in this project, we use the terms “formula” and “parse tree of a formula” interchangeably. This enables us, for example, to say that we traverse the nodes of a formula, when we mean to traverse the nodes of the corresponding AST.

3.1 Abstraction

Our main idea for this first step of the construction of tasks is to transform a given MOSEL formula ϕ into a CHOICE-MOSEL formula ϕ_c, which represents a number of MOSEL formulas. The syntax of CHOICE-MOSEL is defined in figure 1.

In order to define the abstraction of a MOSEL formula, we first consider the type of a node in a MOSEL formula. We note that the syntax of MOSEL as defined in [ADG+13, figure 2] features some primitives, namely string and integer constants, first- and second order quantifiers, boolean operators and integer comparators

MOSEL formulas are made up of a MOSEL-predicate at top level, which is represented by the non-terminal ϕ in the MOSEL-grammar. This predicate may in turn refer to positions and sets, represented

$$
\begin{array}{lll}
n \rightarrow (0\dots 9)^* & c \rightarrow (a\dots z) & str \rightarrow (a\dots z)^* \\
CInt \rightarrow \texttt{ChoiceInt}(n) & CChar \rightarrow \texttt{ChoiceChar}(c) & CStr \rightarrow \texttt{ChoiceString}(s)
\end{array}
$$

$$
\begin{array}{rl}
CPred \rightarrow & \texttt{simBoolOp}(CPred, CPred) \mid \texttt{comBoolOp}(CPred, CPred) \mid \texttt{FOQuant}(str, CPred) \mid \texttt{SOQuant}(str, CPred) \mid \\
& \texttt{neg}(CPred) \mid \texttt{boolConst}() \mid \texttt{posComp}(CPos, CPos) \mid \texttt{atPos}(CChar, CPos) \mid \texttt{atSet}(CChar, CSet) \mid \\
& \texttt{in}(CPos, CSet) \mid \texttt{setCard}(CSet, CInt) \mid \texttt{setCardMod}(CSet, CInt, CInt) \mid \texttt{startEnd}(CStr) \mid \texttt{isEmpty}() \\
CPos \rightarrow & \texttt{FOVar}(str) \mid \texttt{endPos}() \mid \texttt{incDec}(CPos) \mid \texttt{occ}(CStr) \\
CSet \rightarrow & \texttt{SOVar}(str) \mid \texttt{indOf}(CStr) \mid \texttt{setOp}(CSet, CSet) \mid \texttt{all}() \mid \texttt{posComp}(CSet)
\end{array}
$$

Figure 1: The definition of CHOICE-MOSEL

by the nonterminals P and S, respectively. Thus, we define the set of primitive types as $primTyp :=$ $\{$**int**, **string**, **FOquant**, **SOquant**, **boolOp**, **intComp**, **pred**, **pos**, **set**$\}$. We can then define the type of a node as the tuple of types of its arguments. For example, the type of the node $(|S|\%m \text{ CMP } n)$ would be (**set**, **int**, **intComp**, **int**), whereas the type of $(\phi \text{ C } \phi)$ would be (**pred**, **boolOp**, **pred**).

Our abstraction is guided by this idea of types. Two MOSEL-nodes can only be represented by the same CHOICE-MOSEL-node if they have the same type. The converse, however, is not true. Consider, for example, the two formulas $\phi_1 \vee \phi_2$ and $\phi_1 \Rightarrow \phi_2$. If we abstracted these formulas with the same node, it would mean that we could not distinguish between $\phi_1 \Rightarrow \phi_2$ and $\phi_1 \vee \phi_2$ after the abstraction anymore. Constructing the minimal DFA for the former formula requires the student to understand both the concepts of negation as well as the union of automata. The construction of the minimal DFA for the latter formula requires only the concept of intersection, however. Thus, we map these two nodes to different nodes in the abstraction.

$\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$ are abstracted with the same node, however, since they require the concept of intersection and union, however, which are very similar to each other and can thus be interchanged. The transformation function $abstract$: MOSEL $\rightarrow$ CHOICE-MOSEL is defined in figure 2.

$$
\begin{array}{lll}
abs(n) \mapsto CInt(n) & abs(c) \mapsto CChar(c) & abs(s) \mapsto CStr(s)
\end{array}
$$

$$
\begin{array}{ll|ll}
\left.\begin{matrix}\phi_1 \wedge \phi_2 \\ \phi_1 \vee \phi_2\end{matrix}\right\} & \mapsto \texttt{simBoolOp}(abs(\phi_1), abs(\phi_2)) & x & \mapsto \texttt{FOVar}(x) \\
\left.\begin{matrix}\phi_1 \Rightarrow \phi_2 \\ \phi_1 \Leftrightarrow \phi_2\end{matrix}\right\} & \mapsto \texttt{comBoolOp}(abs(\phi_1), abs(\phi_2)) & \left.\begin{matrix}fst \\ last\end{matrix}\right\} & \mapsto \texttt{endPos}() \\
\neg\phi & \mapsto \texttt{neg}(abs(\phi)) & \left.\begin{matrix}P+1 \\ P-1\end{matrix}\right\} & \mapsto \texttt{incDec}(abs(P)) \\
\left.\begin{matrix}\exists x.\phi \\ \forall x.\phi\end{matrix}\right\} & \mapsto \texttt{FOQuant}(x, abs(\phi)) & \left.\begin{matrix}fstOcc(s) \\ lastOcc(s)\end{matrix}\right\} & \mapsto \texttt{occ}(abs(s)) \\
\left.\begin{matrix}true \\ false\end{matrix}\right\} & \mapsto \texttt{boolConst}(x, abs(\phi_1)) & & \\
P_1 \ CMP \ P_2 & \mapsto \texttt{posComp}(abs(P_1), abs(P_2)) & X & \mapsto \texttt{SOVar}(X) \\
a@P & \mapsto \texttt{atPos}(abs(a), abs(P)) & indOf(s)\ \} & \mapsto \texttt{indOf}(abs(s)) \\
a@S & \mapsto \texttt{atSet}(abs(a), abs(S)) & \left.\begin{matrix}S_1 \cap S_2 \\ S_1 \cup S_2\end{matrix}\right\} & \mapsto \texttt{setOp}(abs(S_1), abs(S_2)) \\
P \in S & \mapsto \texttt{in}(abs(P), abs(S)) & & \\
|S|\%m \ CMP \ n & \mapsto \texttt{setCardMod}(abs(S), abs(m), abs(n)) & all\ \} & \mapsto \texttt{all}() \\
|S| \ CMP \ n & \mapsto \texttt{setCard}(abs(S), abs(n)) & \left.\begin{matrix}psLt(P) \\ psLe(P) \\ psGt(P) \\ psGe(P)\end{matrix}\right\} & \mapsto \texttt{posComp}(abs(P)) \\
\left.\begin{matrix}begWt(s) \\ endWt(s)\end{matrix}\right\} & \mapsto \texttt{startEnd}(abs(s)) & & \\
isEmpty & \mapsto \texttt{isEmpty}(abs(S), abs(n)) & &
\end{array}
$$

Figure 2: The definition of abs

3.2 Concretization

In this step our goal is to reverse the abstraction; We are given a CHOICE-MOSEL formula ϕ_c and want to construct the set of all MOSEL formulas ϕ that can be abstracted to ϕ_c. More formally, we want to construct the set $Conc(\phi_c) := \{\phi \mid abs(\phi) = \phi_c\}$.

3.2.1 Idea

The basic idea of concretization is very simple. It is simple to reverse the abstraction by walking over the tree in postorder. If the current node is a leaf, we return all nodes that can be abstracted to this node. Due to our construction of abs, we know that all nodes that can be abstracted to a leaf node are leaf nodes themselves. Thus, we can simply construct them without arguments.

This is not the case for literals, since all integer- and string literals are abstracted to the same node `CInt` and `CStr`, respectively. Thus, the set of concretizations of any node of type `CInt` and `CStr` is infinitely large. However, since we are eventually only interested in the MOSEL formulas whose corresponding minimal DFA is of similar complexity to the minimal DFA for the original formula, we restrict the concretization of these two nodes as follows: `CInt(n)` is concretized to all integers m in the interval $[\max(0, \lfloor m \cdot (1-p) \rfloor), \lceil m \cdot (1+p) \rceil]$ for some p. In our implementation we chose $p = 0.5$. `CStr(s)` is concretized to all strings of the same length as s. This is based on the observation that changing the length of a string constant in a MOSEL formula usually changes the complexity of the corresponding automaton quite drastically.

If the node is an inner node of the formula, we have already concretized all of its arguments. Due to our constraint that two MOSEL nodes can only be abstracted to the same CHOICE-MOSEL node if they have the same type, we know that all possible concretizations take the same number and type of arguments. Thus, we have concretized all possible arguments for all possible concretizations of the current node, which enables us to use them in the concretization of the parent node.

Note that this concretization uses exponential space, since we construct all subtrees recursively. In order to avoid this, we now describe an alternative method of concretization that relies on SMT-constraints, which also allows us greater control over the concretization.

3.2.2 Constraint Generation

The alternative idea for the concretization of a CHOICE-MOSEL-formula ϕ_c is to convert ϕ_c into SMT-constraints, such that each model of these constraints corresponds to one concretization of ϕ_c. We then use a SMT-solver to enumerate all models and reconstruct the corresponding MOSEL-formula for each model. This allows us to enumerate all concretizations using less memory and also enables us to abort the enumeration of concretizations at any point. Furthermore, we are able to impose additional constraints on the resulting formulas. This in turn allows us to exclude formulas whose minimal automaton differs too much in complexity from the original automaton already at this stage.

The general construction of the constraints is shown in figure 3. The function `Number-of-concretizations(CNode)` returns the number of MOSEL-nodes that can be abstracted to the given CHOICE-MOSEL type. Note that the results of this function for all CHOICE-MOSEL-nodes can be precomputed using the definition of abs from figure 2. For example `Number-of-concretizations(comBoolOp)` would be 2, since there are two MOSEL-nodes that can be abstracted into `comBoolOp`.

The transformation from a model to a MOSEL-formula can then be easily performed using a postorder traversal of the CHOICE-MOSEL-formula, where each visited node constructs the MOSEL-node associated

```
function CONC(CHOICE-MOSEL-formula $\phi_c = \texttt{CNode}(arg_1, \ldots, arg_n)$)
    $x \leftarrow$ Fresh-variable
    $numConc \leftarrow$ Number-of-concretizations(CNode)
    $C \leftarrow \emptyset$
    $C \leftarrow C \cup \{0 \leq x, x \leq numConc\}$
    for all $1 \leq i \leq n$ do
        $C \leftarrow C \cup conc(arg_i)$
    end for
end function
```

Figure 3: Algorithm for construction of SMT-constraints from a CHOICE-MOSEL-formula

with the value of its unique variable.

There are two additional constraints on the concretization of constants we can now impose easily. First, we observe that, in order to preserve the concepts used in constructing the minimal DFA for the original formula, it proved to be beneficial to preserve equality of literals. If, for example, a string literal s appears at two places in the original formula, we do not want two different string literals to be concretized at these two places, but we want to use the same literal s' at both places. The same holds for integer literals. This can easily be achieved by reusing the same variables for equal literals instead of getting a fresh variable for every node in the first step of CONC.

Another observation is that is beneficial to also preserve inequality of literals. Consider, for example the formula $begWt(a) \wedge endWt(b)$, i.e., the formula describing the language of all strings that start with an a and end with a b. When constructing the automaton for this formula, the student does not have to consider words of length 1, since no word of this length can start and end with different symbols. If we were to concretize the formula $begWt(a) \wedge endWt(a)$, the student would have to consider this new corner case, which would make the task of constructing the automaton harder. This constraint can easily be imposed by collecting all variables that are used by nodes of type `CInt`, `CChar` and `CStr`, respectively, and impose the additional constraint $x_i \neq x_j$ for each pair of these variables.

These two additional constraints of preserving equality and inequality are not an integral part of our technique. They exist merely as an optimization that allows us to filter out undesirable concretizations already at this early stage which in turn improves the runtime of the complete algorithm. These optimizations might also prevent desirable MOSEL-formulas from being concretized in some cases. Our experiments have shown, however, that the benefit of a shorter runtime greatly outweighs the downside and that the algorithm still produces a sufficient number of good MOSEL-formulas.

3.3 Filtering

We have shown how to produce a set of MOSEL-formulas that are similar in structure to given MOSEL-formula. However, once we finish abstraction and concretization, there are still some formulas that we do not want to give to a student, either because their corresponding automaton is too simple, or because the automaton differs too much from the original automaton. In order to remove these formulas from the output, we filter the result of the concretization step using two different filters. The first one removes those formulas from the output whose automaton is probably too simple to construct. The second one removes those whose automaton differs too much from the original automaton. In order to keep the runtime of these filters low, both filters work heuristically.

3.3.1 Triviality Filter

Since we only manipulate the MOSEL-formulas syntactically, we end up with unsatisfiable formulas in many cases. Consider again the formula $begWt(a) \wedge endWt(b)$. The concretization results, among others, in the formulas $begWt(a) \wedge begWt(b)$ and $endWt(a) \wedge endWt(b)$, both of which are unsatisfiable, i.e., they describe the empty language. In order to exactly recognize this kind of formulas, we would have to construct the automaton for each formula and then check if it consists only of a single state.

Since the construction of the DFA is expensive in terms of runtime, we instead use a method similar to the estimation of the language density in [ADG$^+$13]: We evaluate the concretized MOSEL-formula on all strings up to a certain length. If the formula evaluates to false on all these strings, we assume that it evaluates to false on all strings, i.e., that it describes the empty language, and remove it from the output. With the same reasoning we also remove all formulas that evaluate to true on all strings up to a certain length from the output. In our experiments, we tested each formula on all strings up to length 4, which proved to be a good trade-off between runtime and effectiveness of the filter.

3.3.2 Complexity Filter

After we filter out the trivial formulas, the output might still contain some formulas whose minimal automaton differs too much from the minimal automaton of the original formula. In order to assess the difference between the automaton of the generated formula and the automaton of the original formula correctly, we would have to construct both automata and then compute the minimal edit distance between them, as detailed in [ADG$^+$13]. Since the computation of the minimal edit distance is a computationally very expensive operation in our implementation, we instead only compare the number of states of both automata. If the number of states differs by more than some percentage p, then we reject the formula. In our experiments, we chose $p = 0.2$, which showed to be a good trade-off between rejecting as many undesirable formulas and keeping as many formulas as possible.

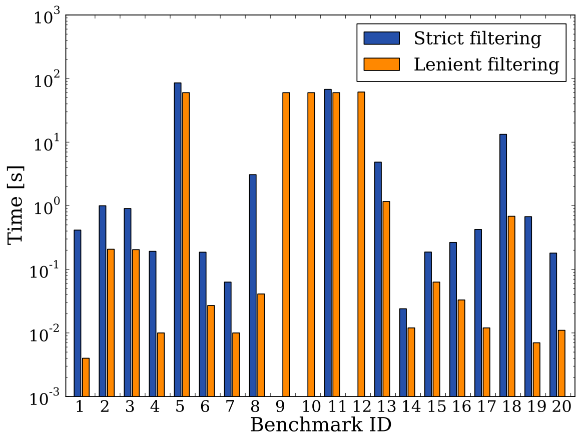
4 Results

We implemented our method using the tool available at `www.automatatutor.com` and tested it using 20 examples of regular languages taken from [Hop01]. In these experiments we chose to preserve both equality and inequality of literals as described in section 3.2.2. We used two different filter configurations. In the first configuration, which we call "lenient filtering", we only applied the triviality filter described in section 3.3.1. In the second configuration, we applied this filter first, and subsequently applied the complexity filter from section 3.3.2. The results of these benchmarks are shown in figure 4.

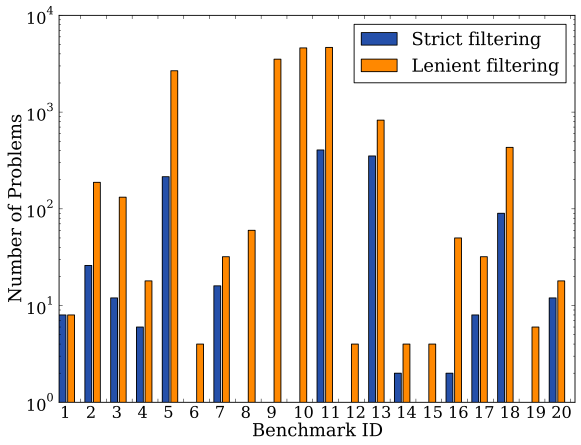
We see that we were able to concretize all formulas in less than 10 seconds for most examples. The outliers are mostly languages taken from a task of the form "Construct a NFA that recognizes the regular language L". This leads to a formula of higher complexity, which in turn slows the algorithm down. We can also observe that the number of generated problems is sufficiently high for most of the benchmarks. The few cases in which no formulas were generated were for exercises for the construction of NFA instead of DFA again.

We have also investigated the relation between the number of variables used in the SMT constraints and the runtime of the complete algorithm. The results are presented in figure 5. Note the logarithmic scale on the y-axis of this figure.

We can see that the runtime of the algorithm increases rapidly if we use more variables in the SMT query, i.e., if we have a larger input formula. The increase in runtime does, however, not appear to be exponential



(a) Runtime of the algorithm



(b) Number of problems generated

Figure 4: Performance of the algorithm on benchmarks

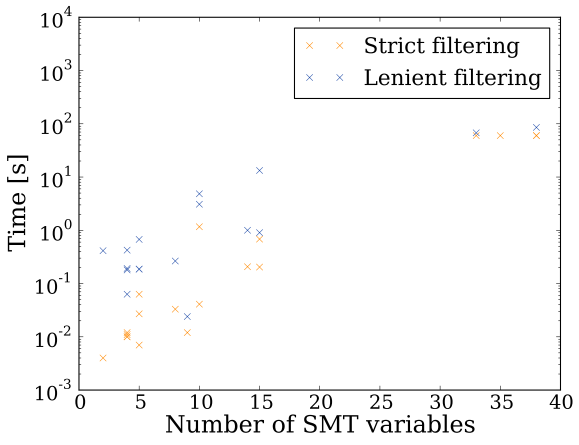


Figure 5: Relation between the runtime and the number of SMT variables used

in the number of variables. This is owed to the fact that the SMT query is of a very simple form, since there are no interdependencies between the constraints and all constraints are of the form $var \leq const$.

5 Conclusions and Future Work

We have presented an algorithm that takes a formal description of a regular language as input and returns a set of regular languages for which the construction of the minimal DFA is of similar difficulty. We have also presented an evaluation of this technique on multiple examples of regular languages taken from one of the most well-known textbooks on automata theory. This evaluation showed that our algorithm produces a sufficient number of regular languages in a reasonably long time that allows for direct interaction with the tool instead of overnight computation.

There are multiple viable directions for future work on this topic. The results in section 4 show that the algorithm produces a sufficient number of new problems in a reasonable time, but they state nothing about the quality of the resulting problems. Even though we have inspected the problems manually and found them to be of similar difficulty as the original problem, a more rigorous investigation should define an objective metric of quality of the results and measure it. One possible approach would be to compare the resulting problems with the original one using the grading metrics from [ADG+13].

Another challenge is that the evaluation shows that, while our algorithm works well on DFA, it times out when constructing new problems of the form “Construct a NFA that recognizes the language L”. This is owed to the fact that, since MOSEL can be seen as a deterministic description language, the mere formulation of these problems in MOSEL transforms the nondeterministic choice into an enumeration of all possibilities. Take, for example, the language of all strings that start and end with the same letter. The MOSEL description for this language is $(begWt(a) \wedge endWt(a)) \vee (begWt(b) \wedge endWt(b))$ This also leads to the problem that the resulting problems are neither well suited for DFA construction nor for NFA construction. However, this could be alleviated by the introduction of nondeterministic operators into MOSEL, which would be an interesting direction for further investigation.

Finally, we approached problem generation using the logical description of a regular language. There are, however, multiple ways to represent such a language. It might feel more natural to manipulate the DFA corresponding to a regular language directly and generate new problems that way. One problem that poses itself, however, would be to ensure that this mutation does not change the concepts that are used for constructing that DFA. Furthermore, a minor change to a minimal automaton might result in an automaton that can be minimized quite drastically, resulting in a construction task of greatly different difficulty. Thus, the generation of tasks using manipulation of DFA would have to be investigated as well.

References

[ADG+13] Rajeev Alur, Loris D’Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. Automated grading of DFA constructions. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, pages 1976–1982, 2013.

[AGK13] Umair Z. Ahmed, Sumit Gulwani, and Amey Karkare. Automatically Generating Problems and Solutions for Natural Deduction. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, pages 1968–1975, 2013.

[Hop01] John Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Pearson Addison Wesley, 2001.

[SGR12] Rohit Singh, Sumit Gulwani, and Sriram K Rajamani. Automatically Generating Algebra Problems. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, pages 1620–1627, 2012.

[SGSL13] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26. ACM, 2013.

[SSG12] Dorsa Sadigh, Sanjit A. Seshia, and Mona Gupta. Automating Exercise Generation: A Step towards Meeting the MOOC Challenge for Embedded Systems. In *Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education*, pages 2:1–2:8, 2012.

[Tho97] Wolfgang Thomas. Languages, automata, and logic. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, pages 389–455. Springer Berlin Heidelberg, 1997.