Class Projects
Due: Project Outline on October 10

Files provided: project-outline.pdf, project-outline.tex
Deliverables: project-outline.pdf, via TigerFile project-outline

As part of this course, students are required to work on a class project. This would typically involve applying a tool to some challenging case study or a new application of your interest, or implementing improved algorithms and methods covered in the course.

The schedule for project-related deliverables is as follows:

- Project outline due: Thursday, October 10
- Project presentations: November 25, December 2, 4 (during lecture time, schedule TBD)
- Project final report: Friday, December 13, 2024 (Dean's date)

We are providing a list of suggested projects, based on topics. You are free to pick one of these, or pick one on your own *after* discussing it with us. We recommend that projects be done in groups of two. This will allow you to divide the work and get more done in your project. (If working in a group, only one student needs to submit the project outline.)

1 Project Categories

Here are some ideas for fun class projects.

Apply automated reasoning in your research. An ideal project is to apply techniques from automated reasoning to a problem in your own field. For example, you might take an intractable problem from your field, reduce it to SAT or SMT, and run experiments to assess the effectiveness of your reduction. You could verify the correctness of an algorithm you are working on using Dafny (an input language and a verifier). If you are interested in distributed systems, you can use IVy, a semi-automated tool that requires user interaction. (Pointers to all these tools are listed at the end.)

Help future COS516 students. Choose a subject that was taught in class and design a tool that helps students explore that subject. For example, you might implement a CDCL solver in a web application that illustrates all of the steps that it takes graphically and allows the user to direct the choices that it makes (e.g., which decision literal to choose, or which cut to make in the implication graph to learn a conflict clause).

Project from suggested list. You can choose one of the projects outlined in the next section. These are organized by topic: building your own program verifier, applying equality saturation for program optimization, verification of a distributed systems protocol, and verification of neural networks.

Please feel free to restrict/extend the scope or consider other variations. For each project, we have provided some pointers and suggestions, to help you define and scope out your project at this early stage. It would be totally fine for you to change/refine the project goals and activities as you make progress during the semester.

2 Project Suggestions

2.1 Build your own program verifier

You may choose to build your own program verifier for a programming language you use frequently. For example, you may implement a Hoare Logic-based Dafny-like verifier for a subset of Python. (We will soon discuss Hoare Logic in class.) You are welcomed to include any additional programming constructs that are related to use cases in your research. You can also choose to design your own programming language and implement a verifier for it.

P1: If you plan to implement a verifier for a subset of a standard programming language, we suggest the following steps:

- Define the syntax of the subset of the language. For instance, if you choose Python, you may want to verify some basic programming constructs such as conditionals, loops and basic arithmetic operations.
- Think about how to obtain the Abstract Syntax Tree (AST) of the grammar you define given some input source code. You may want to use a parser generator such as ANTLR to generate the AST. If there are existing built-in libraries (e.g., the `ast` module of Python), we would highly recommend using available tools.
- Design a way to annotate your program with pre/post-conditions and then implement the verifier using, e.g., weakest preconditions. For instance, if you choose to build a library for Python, you may choose to use decorators to supply pre/post-conditions to functions you are verifying. You can then integrate your tool with a theorem prover such as Z3 to prove the verification conditions.

2.2 Apply Equality Saturation to optimize your programs

Equality saturation (Eqsat) is a powerful tool for optimizing programs – as we have discussed in class (Lecture 6). Unlike traditional term rewriting-based optimizers, Eqsat optimizes programs by memoizing equivalent programs, using a data structure called e-graph (a more powerful congruence closure), given a set of syntactic rewrite rules. These features enable Eqsat to efficiently explore the space of equivalent programs.

The state-of-the-art implementation of Eqsat is `egg` (Willsey et al. 2021). This framework supports defining your own dataflow-based domain-specific language and provides interfaces to define rewrite rules. `egg` also implements the e-graph data structure and automatically converts your DSL programs into e-graphs.

You may choose to apply Eqsat to your domain of interest, to perform automated program optimization. Check out the following pointers to some example applications.

- Optimizing machine learning models. Eqsat has been applied to improve inference speed for machine learning models (Yang et al. 2021). Machine learning models are represented as dataflow-based matrix operations: the input to a layer comes from some previous layer(s); these workloads can be aggressively optimized using Eqsat by exploiting properties of matrix operations.
- Others:
  - Smith et al. 2021: Hardware synthesis for deep learning accelerators.
  - Cao et al. 2022: Library Learning
  - VanHattum et al.: Auto-vectorization/synthesis for DSP
  - Huang et al. Application-level validation of accelerator designs
  - Wang et al. Optimization for database queries

If you are interested in using Eqsat, we encourage you to take a look at one (or some) of the above papers to get a sense of how Eqsat has been used in practical applications.

P2: Use `egg` to optimize programs. If you decide to apply Eqsat to solve problems in your domain of interest, we suggest the following steps:

- Read the `egg` paper and play around with the `egg` library, which will be used to as the Eqsat engine in your project.
- Design a program representation: Since `egg` explores the space of equivalent programs, you will need to design a program representation for solving problems in your domain. For example, in the domain of machine learning, a commonly used representation of machine learning models is the computation graph, which can be programmatically represented as a dataflow graph. This step is crucial because: (1) representations that can be easily defined and handled by `egg` will lower the engineering overhead; (2) an expressive and general representation will enable you to design more interesting evaluation cases.
- Design syntactic rewrite rules: These are custom rewrite rules for your language/representation. For instance, the rewrite rule for algebraic simplifications could be

  $$x \cdot 2 \rightarrow x << 1$$

  These rules can also be some properties your representation satisfies. For example, if your representation involves matrix operations, then you might encode the associativity of matrix multiplication in your rewrite rules. Generally, you need to explore and exploit the properties of your representation when encoding the rewrite rules, which `egg` will apply to your program and populate the e-graph.
- Design extraction heuristics. e-graph facilitates the exploration of the space of equivalent programs, but finally, we will want a canonical (or optimal) representation. This is called *term extraction* in e-graphs. You will need to design a cost model for your program representation and implement an extractor. A simple heuristic could be the size of the program (e.g., the size of the abstract syntax tree). It could be finer-grained, e.g., number of cycles taken by the instructions.
- (optional) Prove the correctness of your rewrite rules (e.g., with an SMT solver). This will help prove the correctness of all the intermediate transformations applied to the e-graph and will ensure the equivalence between the input and the extracted program.

2.3 Verification of Distributed System Protocols

P3: Formal Modeling and Analysis of distributed protocols using P

P is a framework for formally modeling and analyzing various distributed protocols. In this project, you will use P to model distributed protocols of your interests and perform verification using P tools.

Here are some relevant resources for P:

- Getting started on P: https://p-org.github.io/P/
- VSCode Plugin for P: https://p-org.github.io/peasy-ide-vscode/

To begin with, we suggest reading through a tutorial for P, which contains a walk-through of client-server and Two-phase commit protocols (COS 418 slides for Two-phase commit). We provide a list of well-known protocols that are good options to work on, but feel free to pick any protocol that you would like to model and check for correctness.

- Chain replication: high-throughput fault-tolerant data replication.

- Paxos or Raft: consensus protocol.

Note that you may choose to only model the core features of the protocols (e.g., Raft has some mechanism to deal with reconfiguration, which is quite complicated and you can skip modeling that part).

2.4 Large Language Models for Verification and Invariant Generation

Correctness properties of software or hardware systems can be proved at the design stage, to ensure that things do not go wrong when the system is actually deployed in production. Therefore, after the system design is available, a verification expert would like to formally prove that the correctness properties hold for the given system. In practice, the proof of the properties is done by using *induction* on the transition relation of the system (see Lecture 7), where the expert manually provides additional *inductive invariants*, if the properties themselves are not inductive. However, finding such inductive invariants is highly non-trivial and time-consuming.

Recently, large language models (LLMs) are becoming more powerful in logical reasoning and various tasks in software and hardware development. In this project, you will explore how well state-of-the-art LLMs perform when they are prompted with invariant finding tasks.

P4: Evaluating performance of LLMs on finding inductive invariants for distributed systems

The following steps might be helpful for you to formulate and get started to try out various tools:

- Get familiar with IVy. You may not need to know all details of how distributed systems are modeled in Ivy, but you will need to become familiar with how to check whether a given set of invariants is inductive on a given distributed protocol modeled in Ivy. Some examples are included in IVy GitHub repository.
- Choose the LLMs you want to evaluate.
- Take a set of IVy models. We recommend the examples in the IVy GitHub repository and benchmarks in SWISS.
- Design your prompt and evaluate how well the LLMs perform.
- Additionally, you may want to try fine-tuning or prompt engineering the LLMs and see whether you could make them better at finding invariants!

P5: Evaluate performance of LLMs on generating correctness properties for hardware Verilog designs

If you are already familiar with (or are interested in) hardware verification, you can consider exploring the use of LLMs for generating SVA (System Verilog Assertions) that are used to specify correctness properties for Verilog RTL designs. It would be useful to validate the generated assertions using a hardware model checker.

You may want to work on your own favorite Verilog designs and/or hardware model checker. If not, we will be glad to provide you pointers to open-source Verilog designs and a commercial industry-strength model checker (JasperGold), which is available on research computing platforms (Nobel and Adroit) with access via your netid. (We will provide instructions on how to use JasperGold on the cluster.)

2.5 Security Verification for Hardware Designs

The *secure information flow problem* checks that information from high-security inputs (such as passwords) does not leak to any low-security outputs of a program. A well-known approach for verifying secure information flow is based on the notion of self-composition, due to Barthe et al. [CSFW 2004], where the problem is reduced to checking a safety property (i.e., a usual assertion) on *two copies* of the program.

Another popular approach is based on *taint analysis*, where a taint tag is added to program variables, and it is checked whether a tag on the high-security inputs can *propagate* to the low-security outputs. Depending on the taint propagation rules, and precision of the analysis on the program graph, the result may be inaccurate. However, the advantage is that the analysis is done on only a single copy of the program, so it often scales better than self-composition.

(If you like, you can read about these two approaches in one of our papers Wang et al. [CAV 2018]. Our method combines both approaches in a lazy way. The related work in this paper has pointers to many classic references on this topic, but you don't have to read them unless you'd like to!)

P6: Detecting timing side channel leaks in hardware Verilog designs.

In this project, you will study these two approaches – taint propagation and self-composition – when applied to hardware Verilog designs. One way for information to leak is through a timing side channel, i.e., the design may have different execution times based on low/high security inputs. Your goal is to study the tradeoffs in accuracy and performance between these approaches for detecting timing side channel leaks. (If you are interested, we can provide you more pointers to references for timing side channel leaks.)

The main steps in the project are:

- Choose some collection of secure and non-secure Verilog designs. (We can provide you some examples if you like.)
- Add taint logic to the design (using Verilog code) and assertion(s) that a taint does not propagate from any high-security input to any low-security output. Check the assertion(s) using a hardware model checker. (We can provide access to JasperGold, a commercial industry-strength model checker for Verilog designs.)
- Construct a self-composition of the given design, i.e., which has two copies, based on a standard well-known construction. (For small designs, it would be easy and OK to construct this by hand.) Add assertions to check the associated safety property. Again, check these assertions using a hardware model checker.
- If you decide to use JasperGold, the tool provides its own "security checker" that you can apply on the design, and compare with your experiments above.
- Compare the accuracy of the results and model checker performance in the cases above.

2.6 Verification of Deep Neural Networks (DNNs)

Deep neural networks have achieved impressive results in machine learning in various domains, e.g., in image classification. However, they suffer from the problem of non-robustness, i.e., slightly perturbed inputs can fool neural networks, thereby resulting in mis-classification. Such perturbed inputs are called *adversarial examples*.

In some efforts, an SMT solver is used to verify robustness of neural networks. Specifically, given a neural network classifier $f : \mathbb{R}^n \rightarrow [0, \mathbf{C}]$ and an input $x \in \mathbb{R}^n$, the goal is to verify that there is no adversarial example in the neighbor space of x (i.e., all examples in a neighbor space of x should lead to the same output as x):

$$\forall x'.||x' - x||_p \leq \epsilon \rightarrow f(x') = f(x),$$

where $||x' - x||_p \leq \epsilon$ considers an input x' that is slightly perturbed from x. For instance, when the input is an image, $||x' - x||_0 \leq 2$ will consider images x' that are 2-pixel different from x. Given a trained neural network and a dataset, the goal is to check robustness of the model on all images in the dataset.

P7: Verifying DNNs using Marabou: If you have ML models and datasets that you are interested in, you can experiment with Marabou, a state-of-the-art framework for verifying deep neural networks using SMT-based solvers. Marabou can answer two types of queries:

- *Robustness* queries: test whether there exists an adversarial point around a given input point that changes the output of the network.
- *Reachability* queries: if inputs are in a given range, is the output guaranteed to be in some safe range?

More information on Marabou: https://github.com/NeuralNetworkVerification/Marabou

3 List of Tools

The following list provides pointers to tools that you might find useful for your project:

- MiniSAT: A well engineered SAT-solver written in C++. The code is clean and the API is easy to follow and to use. Tool resources are available at: http://minisat.se/. An installation is available on courselab machine at `/u/cos516/minisat`.
- Z3: An SMT solver with support for many theories in FOL. It has many APIs (with support for Python, C, C++, Java and .NET). Tool resources are available at https://github.com/Z3Prover/z3/wiki. An installation is available on courselab machine at `/u/cos516/z3`.
- CVC5: Another SMT solver. In addition to supporting many theories, it also generates proofs of correctness. Learn more at https://cvc5.github.io/.
- `egg`: fast and extensible library for equality saturation. Learn more at https://github.com/egraphs-good/egg
- P: a framework for formally modeling and analyzing distributed protocols. https://p-org.github.io/P/
- Dafny: A language and program verifier for functional correctness.
  https://dafny.org/
- IVy: IVy is a tool for specifying, modeling, implementing and verifying distributed protcols. IVy is intended to allow interactive development of protocols and their proofs of correctness and to provide a platform for developing and experimenting with automated proof techniques. In particular, IVy provides interactive visualization of automated proofs, and supports a use model in which the human protocol designer and the automated tool interact to expose errors and prove correctness.
  Tool: https://microsoft.github.io/ivy
  Paper: PLDI 2016 paper
  Video: PLDI 2016 talk
- JasperGold: a commercial industry-strength model checker for Verilog hardware designs. Access to an academic version is available on the research computing platforms (Nobel and Adroit) via your netid. (We will provide pointers on how to access and use JasperGold, but we may not be able to help with debugging any problems.)
- Any other automated reasoning tool that you might like to use – please discuss with us first!