Verification and LLMs

CS 485/698: AI-Assisted SE

Today’s Agenda

- Team meeting (~5 minutes)
  - If you use Claude Code as your main LLM agent, let me know when I’m coming around (it’s relevant for the in-class)
- Lecture on program verification
- In-class activity: try to build “AI Tinder” using lemmafit

Program Verification

Definition: a ***program verifier*** is a static analysis that proves mathematical theorems (usually called ***specifications***) about the run-time behavior of a program

Program Verification

Definition: a ***program verifier*** is a static analysis that proves mathematical theorems (usually called ***specifications***) about the run-time behavior of a program

- note “proves”: a program verifier must be **sound**

Program Verification

Definition: a ***program verifier*** is a static analysis that proves mathematical theorems (usually called ***specifications***) about the run-time behavior of a program

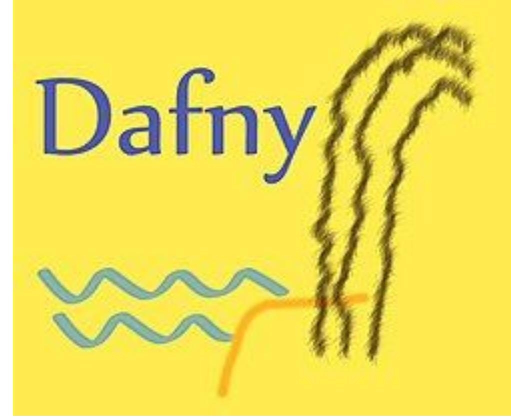
- note "proves": a program verifier must be **sound**
- this definition includes **static type systems**, which we discussed on Monday
  - what are the specifications?

Program Verification

Definition: a ***program verifier*** is a static analysis that proves mathematical theorems (usually called ***specifications***) about the run-time behavior of a program

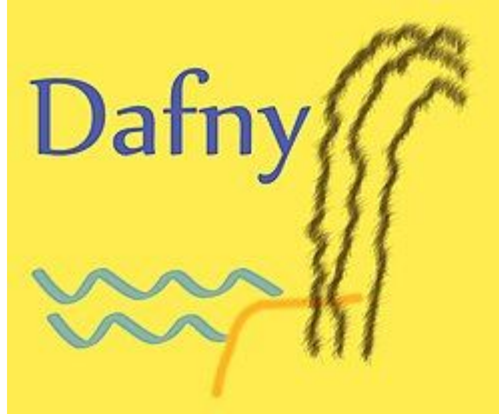
- note “proves”: a program verifier must be **sound**
- this definition includes **static type systems**, which we discussed on Monday
  - what are the specifications?
- today, however, we’re going to focus on verifiers that can prove **arbitrary** (i.e., user-supplied) specifications

Verifier example: Dafny



yes, this is the real logo

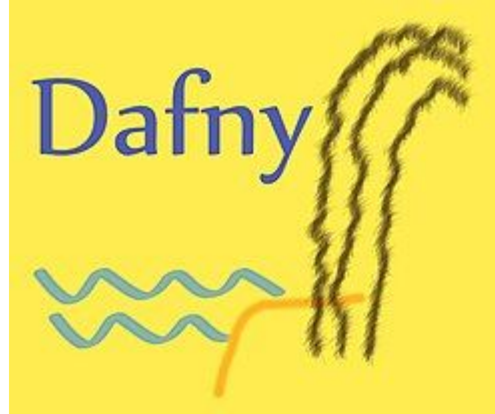
Verifier example: Dafny



yes, this is the real logo

- Dafny is both a **programming language** and a verifier
  - Language is object-oriented; “feels like C# or Java”
  - Dafny docs call it “***verification-aware***”

Verifier example: Dafny



yes, this is the real logo

- Dafny is both a **programming language** and a verifier
  - Language is object-oriented; "feels like C# or Java"
  - Dafny docs call it "***verification-aware***"
- Programmer writes both the program and its specification
  - Dafny attempts to discharge the proof automatically, but the programmer often has to provide hints ("***proof script***")
  - Some specifications (e.g., all array accesses must be in-bounds) are enforced automatically

Dafny Example: Fibonacci

```
function Fib(n: nat): nat {
 if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
{
  x := 0;
  var y, i := 1, 0;
  while i < n
    invariant 0 <= i <= n
    invariant x == Fib(i) && y = Fib(i + 1)
  {
    x, y := y, x + y;
    i := i + 1;
  }
}
```

Dafny Example: Fibonacci

```
function Fib(n: nat): nat {
 if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
{
  x := 0;
  var y, i := 1, 0;
  while i < n
    invariant 0 <= i <= n
    invariant x == Fib(i) && y = Fib(i + 1)
  {
    x, y := y, x + y;
    i := i + 1;
  }
}
```

naive Fibonacci is part of the spec

Dafny Example: Fibonacci

```
function Fib(n: nat): nat {
 if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
{
  x := 0;
  var y, i := 1, 0;
  while i < n
    invariant 0 <= i <= n
    invariant x == Fib(i) && y = Fib(i + 1)
  {
    x, y := y, x + y;
    i := i + 1;
  }
}
```

Dafny has to prove that these produce the same value b/c of “ensures” clause

Dafny Example: Fibonacci

```
function Fib(n: nat): nat {
 if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
{
  x := 0;
  var y, i := 1, 0;
  while i < n
    invariant 0 <= i <= n
    invariant x == Fib(i) && y = Fib(i + 1)
  {
    x, y := y, x + y;
    i := i + 1;
  }
}
```

loop invariants are necessary hints to discharge proof (proof will get stuck without them)

How Dafny works: reduction to SMT

How Dafny works: reduction to SMT

- Under the hood, Dafny is converting your program and specification into a set of ***verification conditions*** - mathematical formulas that encode the properties of interest

How Dafny works: reduction to SMT

- Under the hood, Dafny is converting your program and specification into a set of ***verification conditions*** - mathematical formulas that encode the properties of interest
- These verification conditions are then solved by an ***SMT solver***, an automated tool for formal reasoning

How Dafny works: reduction to SMT

- Under the hood, Dafny is converting your program and specification into a set of ***verification conditions*** - mathematical formulas that encode the properties of interest
- These verification conditions are then solved by an ***SMT solver***, an automated tool for formal reasoning
- As a user of Dafny, ideally this would be transparent to you: you would write your program and spec, and the proof would succeed iff the spec was true

How Dafny works: reduction to SMT

- Under the hood, Dafny is converting your program and specification into a set of ***verification conditions*** - mathematical formulas that encode the properties of interest
- These verification conditions are then solved by an ***SMT solver***, an automated tool for formal reasoning
- As a user of Dafny, ideally this would be transparent to you: you would write your program and spec, and the proof would succeed iff the spec was true
  - In practice, however, it isn't: SMT solvers are unpredictable
    - Does anyone know why?

What is an SMT solver, exactly?

What is an SMT solver, exactly?

Definition: a ***satisfiability-modulo-theories*** (***SMT***) ***solver*** is a tool that tries to automatically either produces a set of assignments to variables in a mathematical formula that makes it true, if such a solution exists; or, if no such solution exists, produces a proof of unsatisfiability.

What is an SMT solver, exactly?

Definition: a ***satisfiability-modulo-theories*** (***SMT***) ***solver*** is a tool that tries to automatically either produces a set of assignments to variables in a mathematical formula that makes it true, if such a solution exists; or, if no such solution exists, produces a proof of unsatisfiability.

- note “tries to”: boolean satisfiability is **NP-complete**

What is an SMT solver, exactly?

Definition: a ***satisfiability-modulo-theories*** (***SMT***) ***solver*** is a tool that tries to automatically either produces a set of assignments to variables in a mathematical formula that makes it true, if such a solution exists; or, if no such solution exists, produces a proof of unsatisfiability.

- note "tries to": boolean satisfiability is **NP-complete**
- "theories" refers to non-boolean parts of the formula
  - for example, a solver might support a **theory of real numbers**

What is an SMT solver, exactly?

Definition: a ***satisfiability-modulo-theories*** (***SMT***) ***solver*** is a tool that tries to automatically either produces a set of assignments to variables in a mathematical formula that makes it true, if such a solution exists; or, if no such solution exists, produces a proof of unsatisfiability.

- note "tries to": boolean satisfiability is **NP-complete**
- "theories" refers to non-boolean parts of the formula
  - for example, a solver might support a **theory of real numbers**
- different solvers might support different theories

What is an SMT solver, exactly?

Definition: a ***satisfiability-modulo-theories*** (***SMT***) ***solver*** is a tool that tries to automatically either produces a set of assignments to variables in a mathematical formula that makes it true, if such a solution exists; or, if no such solution exists, produces a proof of unsatisfiability.

- note “tries to”: boolean satisfiability is **NP-complete**
- “theories” refers to non-boolean parts of the formula
  - for example, a solver might support a **theory of real numbers**
- different solvers might support different theories
- **key idea** for solving verification conditions: use SMT solver to check that they are true
  - but how does it work?

Review: basics of SAT

- You should have seen the ***boolean satisfiability problem*** (***SAT problem***) in your undergraduate theory of computation course
    - but just in case you did not…

Review: basics of SAT

- You should have seen the ***boolean satisfiability problem*** (***SAT problem***) in your undergraduate theory of computation course
    - but just in case you did not…

Definition: a ***boolean formula*** is a set of ***boolean variables*** (i.e., symbols that can be either true or false)

Review: basics of SAT

- You should have seen the ***boolean satisfiability problem*** (***SAT problem***) in your undergraduate theory of computation course
  - but just in case you did not…

Definition: a ***boolean formula*** is a set of ***boolean variables*** (i.e., symbols that can be either true or false) connected by the ***boolean operators*** ($\wedge$ for logical and, $\vee$ for logical or, and ¬ for logical negation)

Review: basics of SAT

- You should have seen the ***boolean satisfiability problem*** (***SAT problem***) in your undergraduate theory of computation course
  - but just in case you did not…

Definition: a ***boolean formula*** is a set of ***boolean variables*** (i.e., symbols that can be either true or false) connected by the ***boolean operators*** ($\wedge$ for logical and, $\vee$ for logical or, and ¬ for logical negation)

- a boolean formula is ***satisfiable*** iff there exists an ***assignment*** of the variables to true and false that makes the formula as a whole evaluate to true

Review: basics of SAT

- You sh *SAT*
  problem course
  - but

Definition ., symbols
that can be *operators*
($\land$ for logical and, $\lor$ for logical or, and $\neg$ for logical negation)

- a boolean formula is *satisfiable* iff there exists an ***assignment*** of the variables to true and false that makes the formula as a whole evaluate to true

Example boolean formulas:

- $a \lor b \land \neg c$
- $(P \land Q) \lor (Q \land \neg R)$
- etc.

SAT solving goal: find an assignment

- You can think of an ***assignment*** as a mapping from variables to values

SAT solving goal: find an assignment

- You can think of an ***assignment*** as a mapping from variables to values
- Examples:
  - is $X \lor Y$ satisfiable?

SAT solving goal: find an assignment

- You can think of an ***assignment*** as a mapping from variables to values
- Examples:
  - is $X \vee Y$ satisfiable?
    - yes: X -> true, Y -> false is a satisfying assignment

SAT solving goal: find an assignment

- You can think of an ***assignment*** as a mapping from variables to values
- Examples:
  - is $X \lor Y$ satisfiable?
    - yes: X -> true, Y -> false is a satisfying assignment
  - is $X \land \neg X$ satisfiable?

SAT solving goal: find an assignment

- You can think of an ***assignment*** as a mapping from variables to values
- Examples:
    - is $X \vee Y$ satisfiable?
        - yes: X -> true, Y -> false is a satisfying assignment
    - is $X \wedge \neg X$ satisfiable?
        - no: there is no choice of X that makes both X and $\neg X$ true at the same time

SAT to SMT: adding in theories

- SMT solvers extend SAT to allow reasoning about constructions other than boolean formulas via ***theories***
  - e.g., maybe we have a verification condition like “x > 10”
  - for this, we’d use the ***theory of linear arithmetic***
    - which has its own dedicated solver

SAT to SMT: adding in theories

- SMT solvers extend SAT to allow reasoning about constructions other than boolean formulas via ***theories***
  - e.g., maybe we have a verification condition like “x > 10”
  - for this, we’d use the ***theory of linear arithmetic***
    - which has its own dedicated solver
- SMT solving trick:
  - replace “x > 10” wherever it appears with a **new boolean variable** (call it α)

SAT to SMT: adding in theories

- SMT solvers extend SAT to allow reasoning about constructions other than boolean formulas via ***theories***
  - e.g., maybe we have a verification condition like “x > 10”
  - for this, we’d use the ***theory of linear arithmetic***
    - which has its own dedicated solver
- SMT solving trick:
  - replace “x > 10” wherever it appears with a **new boolean variable** (call it α)
  - find a satisfying assignment. If α is part of the satisfying assignment, ask the theory solver if α **and anything else in that theory** can be true at the same time

SAT to SMT: adding in theories

- SMT so ... structions
  other t
  - e.g. ... > 10”
  - for
    - ■
- SMT so
  - rep ... **olean var**
  - find ... ying
    assi ... **g else in**
    that theory can be true at the same time

Example:

- formula: $x > 10 \land x < 7$

SAT to SMT: adding in theories

- SMT so… …tructions other t…
  - e.g. … > 10”
  - for…
    - ■
- SMT so…
  - rep… **…olean var…**
  - find… …ying assi… **…g else in that theory** can be true at the same time

Example:

- formula: $x > 10 \land x < 7$
- transform to: $\alpha \land \beta$

SAT to SMT: adding in theories

Example:

- formula: $x > 10 \land x < 7$
- transform to: $\alpha \land \beta$
- SAT solver says α=true, β=true

- SMT so … tructions other t…
  - e.g. … > 10"
  - for …
- SMT so…
  - rep… **olean var…**
  - find… ying assi… g else in that theory can be true at the same time

SAT to SMT: adding in theories

Example:

- formula: $x > 10 \land x < 7$
- transform to: $\alpha \land \beta$
- SAT solver says α=true, β=true
- ask linear arithmetic solver if $x > 10$ and $x < 7$ can be true at the same time
    - “no”

- SMT so… …structions other t…
    - e.g. … > 10”
    - for…
        - ■
- SMT so…
    - rep… **…olean var…**
    - find… …ying assig… **…g else in that theory** can be true at the same time

SAT to SMT: adding in theories

Example:

- formula: $x > 10 \land x < 7$
- transform to: $\alpha \land \beta$
- SAT solver says α=true, β=true
- ask linear arithmetic solver if $x > 10$ and $x < 7$ can be true at the same time
  - “no”
- add this info to SAT formula:

$$\alpha \land \beta \land \neg(\alpha \land \beta)$$

SAT to SMT: adding in theories

Example:

- formula: $x > 10 \land x < 7$
- transform to: $\alpha \land \beta$
- SAT solver says α=true, β=true
- ask linear arithmetic solver if $x > 10$ and $x < 7$ can be true at the same time
  - “no”
- add this info to SAT formula:

  $\alpha \land \beta \land \neg(\alpha \land \beta)$
- SAT solver: unsat!

SMT Solver in Verification

- Modern SMT solvers use **sophisticated heuristics** to solve common patterns in formulas that arise in program verification
  - they can solve (some) formulas with millions or billions of clauses very quickly (under 30 seconds)

SMT Solver in Verification

- Modern SMT solvers use **sophisticated heuristics** to solve common patterns in formulas that arise in program verification
  - they can solve (some) formulas with millions or billions of clauses very quickly (under 30 seconds)
- However, **worst case** is still $O(2^n)$ time for a formula with *n* variables
  - Very slow!

SMT Solver in Verification

- Modern SMT solvers use **sophisticated heuristics** to solve common patterns in formulas that arise in program verification
  - they can solve (some) formulas with millions or billions of clauses very quickly (under 30 seconds)
- However, **worst case** is still $O(2^n)$ time for a formula with *n* variables
  - Very slow!
- In practice, it's hard to tell if a program/specification pair will trigger pathological behavior from the SMT solver
  - As a result, verification can be **unpredictable and chaotic**

SMT Solver in Verification

- Modern SMT solvers use **sophisticated heuristics** to solve common … rification
  - they … ons of clau…
- Howeve… h n variable…
  - Very…
- In practi… r will trigger pa…
  - As a result, verification can be **unpredictable and chaotic**

Not all verifiers rely on reduction to SMT. For example, Rocq is type systems all the way down (***calculus of constructions***). However, there is a tradeoff between automation and annotation burden - user must write the whole proof (not just the specs!) in Rocq.

Verifiers in practice: example successes

- CompCert: a verified C compiler [1]
    - Csmith authors: *"The striking thing about our CompCert results is that the middleend bugs we found in all other compilers are absent."* [2]

[1] Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., & Ferdinand, C. *CompCert-a formally verified optimizing compiler.* ERTS 2016
[2] Yang, X., Chen, Y., Eide, E., & Regehr, J. *Finding and understanding bugs in C compilers*. PLDI 2011.

Verifiers in practice: example successes

- CompCert: a verified C compiler [1]
  - Csmith authors: *"The striking thing about our CompCert results is that the middleend bugs we found in all other compilers are absent."* [2]
- Dafny used to prove correctness of AWS authentication engine, encryption SDK
  - https://youtu.be/oshxAJGrwMU?si=2HRf2XvNR-8RMIXZ
  - https://github.com/aws/aws-encryption-sdk

[1] Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., & Ferdinand, C. *CompCert-a formally verified optimizing compiler.* ERTS 2016
[2] Yang, X., Chen, Y., Eide, E., & Regehr, J. *Finding and understanding bugs in C compilers*. PLDI 2011.

Verifiers in practice: downsides

- Verifiers share the **same downsides** as other static analyses:

Verifiers in practice: downsides

- Verifiers share the **same downsides** as other static analyses:
  - too many false alarms
  - too much effort to apply to a codebase

Verifiers in practice: downsides

- Verifiers share the **same downsides** as other static analyses:
  - too many false alarms
  - too much effort to apply to a codebase
- However, in verifiers these problems are **exacerbated**:
  - with automation (e.g., SMT solvers) false alarms can happen for seemingly arbitrary reasons

Verifiers in practice: downsides

- Verifiers share the **same downsides** as other static analyses:
  - too many false alarms
  - too much effort to apply to a codebase
- However, in verifiers these problems are **exacerbated**:
  - with automation (e.g., SMT solvers) false alarms can happen for seemingly arbitrary reasons
  - human effort is significantly higher
    - for example, CompCert proof (only) is ~100,000 lines (https://www.absint.com/compcert/structure.htm)

The future: agentic guess-and-check loop?

- Verifiers are the **strongest static analyzers** possible

The future: agentic guess-and-check loop?

- Verifiers are the **strongest static analyzers** possible
- So, combining them with LLMs in an agentic loop makes sense:

The future: agentic guess-and-check loop?

- Verifiers are the **strongest static analyzers** possible
- So, combining them with LLMs in an agentic loop makes sense:
  - LLM proposes code and specifications

The future: agentic guess-and-check loop?

- Verifiers are the **strongest static analyzers** possible
- So, combining them with LLMs in an agentic loop makes sense:
  - LLM proposes code and specifications
  - Verifier checks that they match

The future: agentic guess-and-check loop?

- Verifiers are the **strongest static analyzers** possible
- So, combining them with LLMs in an agentic loop makes sense:
  - LLM proposes code and specifications
  - Verifier checks that they match
  - Human engineers check the specifications only

The future: agentic guess-and-check loop?

- Verifiers are the **strongest static analyzers** possible
- So, combining them with LLMs in an agentic loop makes sense:
  - LLM proposes code and specifications
  - Verifier checks that they match
  - Human engineers check the specifications only
- Tools that implement this loop are just starting to appear

The future: agentic guess-and-check loop?

- Verifiers are the **strongest static analyzers** possible
- So, combining them with LLMs in an agentic loop makes sense:
  - LLM proposes code and specifications
  - Verifier checks that they match
  - Human engineers check the specifications only
- Tools that implement this loop are just starting to appear
- Today's in-class activity: try out **lemmafit**, a state-of-the-art tool
  - https://lemmafit.com/
  - uses Dafny under the hood as the verifier
  - sadly only works with Claude Code (for now)

In-class activity: AI Tinder with Lemmafit

- Recall the “AI Tinder” spec that we used earlier in the class:
    - Like, Super Like, and Reject actions (frontend and backend)
    - Basic profiles: photos, names, bios (?)
    - Push notifications on match
    - Implement as a webapp
- Each group should write a single prompt together. Then:
    - team member w/ Claude Code: use Lemmafit to “one-shot” it
    - another team member: use a different agentic coding tool
        - but the same prompt!
    - compare and discuss results amongst yourselves

Whole-class Discussion

- How did it go?
- Compare with Lemmafit to without Lemmafit
  - e.g., to what you generated during the frontend/backend lectures earlier in the semester
- Was verification *useful*? Did it catch real problems?
- Did you notice verification running in the background? Does it add *noticeable* latency compared to “regular” AI code generation?
- Other commentary on verification + LLMs?

Wrapup and Reminders

- For grad students: don't forget about A7
  - remember that you have to prepare to lead a discussion on your chosen paper
- Next week's plan:
  - Monday: guest lecture from Michael on evaluating the quality of LLM-generated code
  - Wednesday: A7 discussion
    - undergrads do not need to prepare
- Week of May 4: project presentations
  - make sure you have practiced!