Decision Procedures for Software Verification

Lecture 0: Overview

Christophe Ringeissen

2020/21

Outline

Program Verification and Logical Problems

Many approaches to software verification:

- static program analysis
- software model checking (e.g., BLAST, SLAM)
- abstraction interpretation
- deductive verification (e.g., Why3, OpenJML)

require to discharge some proof obligations, i.e.

- **checking that some formula, usually, of first-order logic with equality (FOLE) is valid w.r.t. (modulo) a given theory modeling**
  - the user-defined data types of the software system under scrutiny
  - the memory model used by the programming language
  - its type system, ...

For such program verification techniques, it is of crucial importance to have *reasoning tools* which are both

- *scalable*
- *predictable*, and
- *expressive*

Scenario 1: what is an (optimizing) compiler?

Definition (Compilers)

Special programs that take instructions written in a high level language (e.g., C, Pascal) and convert it into machine language or code the computer can understand.

Example

Consider the following simple program fragment in C:

```
...
int x,y,z;
s0: ...  /* y and z are initialized */
s1: x = (y+z) * (y+z) * (z+y) * (z+y);
...
```

Problem: sub-expressions are needlessly re-computed*!*

Scenario 1: an (optimizing) compiler

Example (cont'd)

By exploiting only the syntactic structure of sub-expressions, transform

```
int x,y,z;
s0: ...  /* y and z are initialized */
s1: x = (y+z) * (y+z) * (z+y) * (z+y);
```

into

```
int x,y,z; int aux1,aux2;
t0: ...  /* y and z are initialized */
t1: aux1 = (y+z);
t2: aux2 = (z+y);
t3: x = aux1 * aux1 * aux2 * aux2;
```

which avoids the re-computation of sub-expressions!

Correctness of the compiler as a logical problem

Example (cont'd)

QUESTION: how can we guarantee that the value stored in $\mathtt{x}$ after the computation of the transformed program is equal to that in $\mathtt{x}$ after the computation of the source *?*
ANSWER: ignore the arithmetic properties of all arithmetic operations and consider them as uninterpreted functions (UF) (i.e. $+ \rightsquigarrow f$ and $* \rightsquigarrow g$). Then, discharge the following proof obligation:

$$T_{UF} \models \left(\begin{array}{l} y_{s0} = y_{t0} \wedge z_{s0} = z_{t0} \\ x_{s1} = g(g(f(y_{s0}, z_{s0}), f(y_{s0}, z_{s0})), g(f(z_{s0}, y_{s0}), f(z_{s0}, y_{s0}))) \\ aux1_{t1} = f(y_{t0}, z_{t0}) \\ aux2_{t2} = f(z_{t0}, y_{t0}) \\ x_{t3} = g(g(aux1_{t1}, aux1_{t1}), g(aux2_{t2}, aux2_{t2})) \end{array} \right) \Rightarrow x_{s1} = x_{t3}$$

What is T_{UF} *?*

Theory of Uninterpreted Functions (T_{UF})

- The following axioms stipulates that $=$ is a congruence relation (i.e., a reflexive, symmetric, and transitive relation closed under substitution of equals by equals)

$$
\begin{array}{ll}
\forall x. & (x = x) \\
\forall x, y. & (x = y \Rightarrow y = x) \\
\forall x, y, z. & (x = y \wedge y = z \Rightarrow x = z) \\
\forall x_1, \ldots, x_n, y_1, \ldots, y_n. & (x_1 = y_1 \wedge \cdots \wedge x_n = y_n) \\
 & \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \\
 & \text{for each function symbol } f \text{ of arity } n
\end{array}
$$

  T_{UF} is the set of sentences which are true in the class of all structures satisfying the above axioms (i.e., models of T_{UF})
- How do we reason in T_{UF}? In particular, how can we design techniques to reason in T_{UF}?

Scenario 2: a more complex (optimizing) compiler

Example (cont'd)

By exploiting the syntactic structure of sub-expressions and properties of $+$, transform

```
int x,y,z;
s0: ...  /* y and z are initialized */
s1: x = (y+z) * (y+z) * (z+y) * (z+y);
```

into

```
int x,y,z; int aux;
t0: ...  /* y and z are initialized */
t1: aux = (y+z);
t2: x = aux * aux * aux * aux;
```

which avoids the re-computation of sub-expressions!

Revisiting the correctness of the compiler

Example (cont'd)

QUESTION: how can we guarantee that the value stored in $\mathtt{x}$ after the computation of the transformed program is equal to that in $\mathtt{x}$ after the computation of the source*?*
ANSWER: ignore the arithmetic properties of * and consider it as an uninterpreted function g. Then, discharge the following proof obligation:

$$T_{UF} \cup T_{LA} \models \left(\begin{array}{l} y_{s0} = y_{t0} \wedge z_{s0} = z_{t0} \\ x_{s1} = g(g(y_{s0} + z_{s0}, y_{s0} + z_{s0}), g(z_{s0} + y_{s0}, z_{s0} + y_{s0})) \\ aux_{t1} = y_{t0} + z_{t0} \\ x_{t2} = g(g(aux_{t1}, aux_{t1}), g(aux_{t1}, aux_{t1})) \end{array} \right) \Rightarrow x_{s1} = x_{t2}$$

What is T_{LA}*?* What does it mean $T_{UF} \cup T_{LA}$*?* Why can we not ignore the arithmetic properties of +*?*

A definition of T_{LA}, $T_{UF} \cup T_{LA}$, and ...

- T_{LA} the set of all sentences that are true in the structure of the integer numbers interpreting $+$ as addition
- $T_{UF} \cup T_{LA}$ is the union of T_{UF} and T_{LA}, i.e. the set of all sentences that are true in all models of both T_{UF} and T_{LA}
- How do we reason in T_{UF}?
  How do we reason in T_{LA}?
  How do we reason in $T_{UF} \cup T_{LA}$?
  In particular, can we modularly re-use our techniques to reason in T_{UF} and T_{LA} for their union?

Scenario 3: an example of program verification

- ***I/O***: program *Prg* manipulating a square matrix *r* of dimension *n*
- ***What***: *Prg* updates only the elements on the diagonal of *r*
- ***Property***: Check that if the matrix *r* is symmetric before the execution of *Prg*, then the matrix r' (obtained by the execution of *Prg* with *r* as input) is also symmetric

Reduction to a logical problem

- First, we need to formalize the bi-dimensional matrix data structure. For this, we use the theory of arrays (T_{Arr}):

$$\begin{aligned} \forall A, I, E.\text{read}(\text{write}(A, I, E), I) &= E \\ \forall A, I, J, E. I \neq J \Rightarrow \text{read}(\text{write}(A, I, E), J) &= \text{read}(A, J) \end{aligned}$$

- Second, we need pairs (for indexing in a bi-dimensional structure) (T_{Pair}):

$$\begin{aligned} \forall I, J.\pi_1(\langle I, J\rangle) &= I \\ \forall I, J.\pi_2(\langle I, J\rangle) &= J \\ \forall P.\langle \pi_1(P), \pi_2(P)\rangle &= P \end{aligned}$$

- Third, define a symmetric matrix m of dimension n (D_{Symm}):

$$\forall r, n.\text{symm}(r, n) \Leftrightarrow \forall i, j.(0 \leq i, j < n \Rightarrow \text{read}(r, \langle i, j\rangle) = \text{read}(r, \langle j, i\rangle))$$

Reduction to a logical problem (cont'd)

- For dimension $n = 2$, discharge the following proof obligation:

$$T_{Arr} \cup T_{Pair} \cup D_{Symm} \models \left(\begin{array}{l} \text{symm}(r, 2) \wedge \\ r' = \text{write}(\text{write}(r, \langle 0, 0 \rangle, e_0), \langle 1, 1 \rangle, e_1) \end{array} \right) \Rightarrow \text{symm}(r', 2)$$

- The above formula expresses the fact that *Prg* returns a symmetric matrix r' if it takes as input a symmetric matrix r and it changes only the elements on the diagonal of r
- Because of the def. of symm, the formula contains quantifiers!
- How can we handle quantifiers? How can we combine quantifier handling and reasoning in the combination of T_{Arr} and T_{Pair}?

One more remark about the last proof obligation...

By unfolding the definition of symm, we can transform

$$T_{Arr} \cup T_{Pair} \cup D_{Symm} \models \left(\begin{array}{l} \text{symm}(r, 2) \wedge \\ r' = \text{write}(\text{write}(r, \langle 0, 0 \rangle, e_0), \langle 1, 1 \rangle, e_1) \end{array} \right) \Rightarrow \text{symm}(r', 2)$$

into

$$T_{Arr} \cup T_{Pair} \models \left(\begin{array}{l} \forall i, j.(0 \leq i, j < 2 \Rightarrow \text{read}(r, \langle i, j \rangle) = \text{read}(r, \langle j, i \rangle)) \wedge \\ r' = \text{write}(\text{write}(r, \langle 0, 0 \rangle, e_0), \langle 1, 1 \rangle, e_1) \end{array} \right) \Rightarrow \forall i, j.(0 \leq i, j < 2 \Rightarrow \text{read}(r', \langle i, j \rangle) = \text{read}(r', \langle j, i \rangle))$$

which has a simpler background theory but a more complex Boolean structure than before (i.e. it is not an implication of the form $\bigwedge_i l_i \Rightarrow l$)

- How can we efficiently handle Boolean structure of the formula? How can we integrate this with reasoning in the background theory? And then with quantifier handling?

The problem addressed in this lecture

Problem

*Design **reasoning tools** capable of automatically discharging the largest possible number of proof obligations coming from the widest range of program verification problems*

- Needless to say, this task is far from simple for several reasons...

Some observations on background theories

- First-order logic is undecidable and general purpose provers are not adequate to discharge proof obligations arising from verification problems *!*
- Some interesting (for verification) fragments are decidable, e.g. quantifier-free formulae in T_{UF}, T_{Arr}, T_{LA}, or their combinations ...
- Also some fragments with quantifiers are decidable: formulae in T_{LA}, but ...
- Many other fragments with quantifiers become undecidable: formulae in T_{Arr}, but ...
- The restricted use of quantifiers may allow us to get decidability back, e.g., arrays with dimension, sortedness, etc

Moral

Choose your background theory carefully: trade-off between expressiveness (easy to express properties of the program) and predictability (easy to mechanize the program verification problem)

Some observations on Boolean reasoning

- Most of the times, reducing/encoding program verification problems into logical problems generate large proof obligations with a complex Boolean structure
- There exists efficient techniques to reason in background theories without taking into account the Boolean structure of the formula and there exists very efficient solvers (so called, SAT solvers) to handle the Boolean structure of formulae

Moral

Separate reasoning in the background theory and Boolean reasoning and then carefully integrate them (so to scale up significantly)

Some observations on Quantifier handling

- Usually, introducing quantifiers in proof obligations endanger predictability since the logical problem may become undecidable
- Instantiating quantified variables is far from trivial and it has not been solved satisfactorily

Moral

Introduce as few quantifiers as possible, in a controlled way (so to maintain predictability) and integrate these techniques on top of those for reasoning in background theories and for Boolean solving

First-Order Logic (Syntax)

Let Σ be a signature (i.e., a set of function and predicate symbols with their arities) and X a denumerable set of variables

- A Σ-term is built over function symbols in Σ and variables in X
- A Σ-atom is of the form $p(t_1, \ldots, t_m)$ where p is a predicate symbol in Σ and $t_1, \ldots, t_m$ are Σ-terms
- The set of Σ-formulas is the smallest set of formulas including Σ-atoms which is closed by boolean connectives and quantification over variables
- A Σ-formula is ground if it contains no variables
- A Σ-formula is closed if it contains no free variables
- A Σ-theory is a set of closed Σ-formulas

First-Order Logic (Semantics)

- A Σ-structure is given by a non-empty set (the domain) together with an interpretation of each symbol in Σ as a function or a relation of appropriate arity
- Σ-interpretation is a Σ-structure together with an interpretation for the variables (i.e., values for variables)
  ➽ An interpretation $\mathcal{I}$ defines a value $\mathcal{I}[t]$ for any term t and a truth value $\mathcal{I}[\varphi]$ for any formula φ
- Notion of model:
  Given a formula φ, $\mathcal{I} \models \varphi$ if $\mathcal{I}[\varphi]$ is true
  Given a theory T, $\mathcal{I} \models T$ if for any $\varphi \in T$, $\mathcal{I} \models \varphi$

Validity versus Satisfiability

Given a Σ-theory T and a Σ-formula φ

- Validity problem: $T \models \varphi$ if for any $\mathcal{I}$, $\mathcal{I} \models T$ implies $\mathcal{I}[\varphi]$ is true
- Satisfiability problem: φ is T-satisfiable if for some $\mathcal{I}$, $\mathcal{I} \models T$ and $\mathcal{I}[\varphi]$ is true
- Notice that $T \models \varphi$ iff $\neg\varphi$ is T-unsatisfiable

Moral

From now on, wlog, we will consider Satisfiability Modulo Theories (SMT) problems

- Constraint satisfiability problem modulo T: it is an SMT problem where only constraints (i.e. quantifier-free conjunctions of literals) are considered
- Once you can solve a constraint SMT, you can also solve an SMT problem for quantifier-free formulae (Why? Hint: think of Disjunctive Normal Form transformation...)

Satisfiability Procedures

- An SMT problem is decidable if for any formula φ in a given class (e.g., quantifier-free), it is possible to check if φ is satisfiable or not
- A decision procedure for an SMT problem (also called, satisfiability procedure or SMT solver) is an algorithm solving the given SMT problem

A last (important) problem

- The ultimate goal of program verification is to enforce the correctness of programs...
- In order to believe that a program is correct, we should trust the results of our program verification techniques...
- Now, since we encode program verification problems into SMT problems, we should trust the SMT solver used to solve such problems, but...
- ... in general, SMT solvers are complex pieces of software, designed with efficiency in mind rather than correctness...
- So, how can we certify the results of SMT solvers? *Proofs!*
- Notice that proofs can be exported to proof assistants (e.g., Isabelle or Coq) to lessen the burden of interactive theorem proving*!*

The SMT-Library initiative

- Many SMT solvers around but difficulties in comparing their performances: slightly different definitions of background theories, no common input format for benchmarks, ...
- SMT-Lib addresses this difficulties: common format and set of reference background theories
- Initiative backed-up worldwide by all research group around the world
- Have a look at `http://www.smt-lib.org`
- Spin-off of SMT-Lib: SMT-COMPetition (similar to the SAT competition)
- veriT: a SMT solver developed in Nancy, by David Déharbe and Pascal Fontaine, see `http://www.verit-solver.org/`

Lecture Timetable

1. This introduction (done)
2. Decision Procedures for the theory of Equality
3. Decision Procedures and Solvers for Linear Arithmetic
4. Combining Decision Procedures
5. Building Decision Procedures for Data Structures
6. Boolean Solving
7. Integrating Decision Procedures with Boolean Solving
8. Applications: the veriT SMT solver

Underlying theories

1. Some motivating examples
2. $T = UF$
3. $T = LA$
4. $T = T_1 \cup T_2$
5. $T = Arrays, Lists, Records, Trees, \ldots$
6. SAT
7. $SMT = SAT + T$
8. $veriT = SMT$ solver