First-Order Theories & SMT Solvers

Maria João Frade

HASLab - INESC TEC
Departamento de Informática, Universidade do Minho

2023/2024

First-Order Theories

Introduction

- When judging the validity of first-order formulas we are typically interested in a particular domain of discourse, which in addition to a specific underlying vocabulary includes also properties that one expects to hold.

- For instance, in formal methods involving the integers, one is not interested in showing that the formula

$$\forall x, y.\ x < y \to x < y + y$$

  is true for all possible interpretations of the symbols $<$ and $+$, but only for those interpretations in which $<$ is the usual ordering over the integers and $+$ is the addition function.

- We are not interested in validity in general but in validity with respect to some **background theory** – a logical theory that fixes the interpretations of certain predicates and function symbols.

Introduction

- Stated differently, we are often interested in moving away from pure logical validity (i.e. validity in all models) towards a more refined notion of validity restricted to a specific class of models.

- There are two ways for specifying such a class of models:
  - To provide a *set of axioms* (sentences that are expected to hold in them).
  - To pinpoint the models of interest.

- *First-order theories* provide a basis for the kind of reasoning just described.

Theories - basic concepts

Let $\mathcal{V}$ be a vocabulary of a first-order language.

- A *first-order theory* $\mathcal{T}$ is a set of $\mathcal{V}$-sentences that is closed under derivability (i.e., $\mathcal{T} \models \phi$ implies $\phi \in \mathcal{T}$).
- A *$\mathcal{T}$-structure* is a $\mathcal{V}$-structure that validates every formula of $\mathcal{T}$.
- A formula ϕ is *$\mathcal{T}$-valid* if every $\mathcal{T}$-structure validates ϕ.
- A formula ϕ is *$\mathcal{T}$-satisfiable* if some $\mathcal{T}$-structure validates ϕ.
- Two formulae ϕ and ψ are *$\mathcal{T}$-equivalent* if $\mathcal{T} \models \phi \leftrightarrow \psi$ (i.e, for every $\mathcal{T}$-structure $\mathcal{M}$, $\mathcal{M} \models \phi$ iff $\mathcal{M} \models \psi$).

Theories - basic concepts

Let $\mathcal{T}$ be a first-order theory.

- $\mathcal{T}$ is said to be a *consistent* theory if at least one $\mathcal{T}$-structure exists.
- $\mathcal{T}$ is said to be a *complete* theory if, for every $\mathcal{V}$-sentence ϕ, either $\mathcal{T} \models \phi$ or $\mathcal{T} \models \neg\phi$.
- $\mathcal{T}$ is said to be a *decidable* theory if there exists a decision procedure for checking $\mathcal{T}$-validity.
- A subset $\mathcal{A} \subseteq \mathcal{T}$ is called an *axiom set* for the theory $\mathcal{T}$, when $\mathcal{T}$ is the deductive closure of $\mathcal{A}$, i.e. $\phi \in \mathcal{T}$ iff $\mathcal{A} \models \phi$.
- A theory $\mathcal{T}$ is *finitely* (resp. *recursively*) *axiomatizable* if it possesses a finite (resp. recursive) set of axioms.
- A *fragment* of a theory is a syntactically-restricted subset of formulae of the theory.

Theories - some results

- For a given $\mathcal{V}$-structure $\mathcal{M}$, the theory $\mathrm{Th}(\mathcal{M}) = \{\phi \mid \mathcal{M} \models \phi, \text{for all }\}$ is complete.
  - These semantically defined theories are useful when one is interested in reasoning in some specific mathematical domain such as the natural numbers, rational numbers, etc.
  - Such theories may lack an axiomatisation, which seriously compromises its use in purely deductive reasoning.
- If a theory is complete and recursive axiomatizable, it can be shown to be decidable.

Theories - decidability problem

- The decidability criterion for $\mathcal{T}$-validity is crucial for mechanised reasoning in the theory $\mathcal{T}$.
- It may be necessary (or convenient) to restrict the class of formulas under consideration to a suitable *fragment* (i.e., syntactical constraint).
- The $\mathcal{T}$-validity problem in a fragment refers to the decision about whether or not $\phi \in \mathcal{T}$ when ϕ belongs to the fragment under consideration.
- A fragment of interest is the *quantifier-free (QF) fragment*.

Equality and uninterpreted functions $\mathcal{T}_{\mathsf{E}}$

- The **vocabulary** of the theory of *equality* $\mathcal{T}_{\mathsf{E}}$ consists of
  - equality ($=$), which is the only interpreted symbol (whose meaning is defined via the axioms of $\mathcal{T}_{\mathsf{E}}$);
  - constant, function and predicate symbols, which are uninterpreted (except as they relate to $=$).
- **Axioms:**
  - *reflexivity:* $\forall x.\ x = x$
  - *symmetry:* $\forall x, y.\ x = y \rightarrow y = x$
  - *transitivity:* $\forall x, y, z.\ x = y \wedge y = z \rightarrow x = z$
  - *congruence for functions:* for every function $f \in \mathcal{T}$ with $\mathsf{ar}(f) = n$,

    $$\forall \overline{x}, \overline{y}.\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

  - *congruence for predicates:* for every predicate $P \in \mathcal{T}$ with $\mathsf{ar}(P) = n$,

    $$\forall \overline{x}, \overline{y}.\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n) \rightarrow (P(x_1, \ldots, x_n) \leftrightarrow P(y_1, \ldots, y_n))$$

- $\mathcal{T}_{\mathsf{E}}$-validity is undecidable, but efficiently decidable for the QF fragment.

Presburger arithmetic $\mathcal{T}_{\mathbb{N}}$

- The theory of *Presburger arithmetic* $\mathcal{T}_{\mathbb{N}}$ is the additive fragment of the theory of Peano.
- **Vocabulary:** $\mathcal{V}_{\mathbb{N}} = \{0, 1, +, =\}$
- **Axioms:**
  - axioms of $\mathcal{T}_{\mathsf{E}}$
  - $\forall x.\ \neg(x + 1 = 0)$ *(zero)*
  - $\forall x, y.\ x + 1 = y + 1 \rightarrow x = y$ *(successor)*
  - $\forall x.\ x + 0 = x$ *(plus zero)*
  - $\forall x, y.\ x + (y + 1) = (x + y) + 1$ *(plus successor)*
  - for every formula ϕ with $\mathsf{FV}(\phi) = \{x\}$ *(axiom schema of induction)*

    $$\phi[0/x] \wedge (\forall x.\ \phi \rightarrow \phi[x + 1/x]) \rightarrow \forall x.\ \phi$$

- $\mathcal{T}_{\mathbb{N}}$ is both complete and decidable (Presburger, 1929), but it has double exponential complexity.

Linear integer arithmetic $\mathcal{T}_{\mathbb{Z}}$

- **Vocabulary:** $\mathcal{V}_{\mathbb{Z}} = \{\ldots, -2, -1, 0, 1, 2, \ldots, -3\cdot, -2\cdot, 2\cdot, 3\cdot, \ldots, +, -, >, =\}$
- Each symbol is interpreted with its standard mathematical meaning in $\mathbb{Z}$.
  - Note: $\ldots, -3\cdot, -2\cdot, 2\cdot, 3\cdot, \ldots$ are unary functions. For example, the intended meaning of $3 \cdot x$ is $x + x + x$, and of $-2 \cdot x$ is $-x - x$.

$\mathcal{T}_{\mathbb{Z}}$ and $\mathcal{T}_{\mathbb{N}}$ have the same expressiveness

- For every formula of $\mathcal{T}_{\mathbb{Z}}$ there is an equisatisfiable formula of $\mathcal{T}_{\mathbb{N}}$.
- For every formula of $\mathcal{T}_{\mathbb{N}}$ there is an equisatisfiable formula of $\mathcal{T}_{\mathbb{Z}}$.

- $\mathcal{T}_{\mathbb{Z}}$ is both complete and decidable via the rewriting of $\mathcal{T}_{\mathbb{Z}}$-formulae into $\mathcal{T}_{\mathbb{N}}$-formulae.

- One usually works with the theory of Linear Integer Arithmetic (LIA).
- Note that non-linear integer arithmetic is undecidable even for the quantifier free case.

Linear rational arithmetic $\mathcal{T}_{\mathbb{Q}}$

- The full theory of rational numbers (with addition and multiplication) is *undecidable*.
- But the theory of *linear arithmetic over rational numbers* $\mathcal{T}_{\mathbb{Q}}$ is decidable, and actually more efficiently than the corresponding theory of integers.
- **Vocabulary:** $\mathcal{V}_{\mathbb{Q}} = \{0, 1, +, -, =, \geq\}$
- **Axioms:** 10 axioms (see Manna's book)
- Rational coefficients can be expressed in $\mathcal{T}_{\mathbb{Q}}$.

The formula $\frac{5}{2}x + \frac{4}{3}y \leq 6$ can be written as the $\mathcal{T}_{\mathbb{Q}}$-formula

$$36 \geq 15x + 8y$$

- $\mathcal{T}_{\mathbb{Q}}$ is decidable and its quantifier-free fragment is efficiently decidable.

Reals $\mathcal{T}_{\mathbb{R}}$

- Surprisingly, the *theory of reals* $\mathcal{T}_{\mathbb{R}}$ is decidable even in the presence of multiplication and quantifiers.
- **Vocabulary:** $\mathcal{V}_{\mathbb{R}} = \{0, 1, +, \times, -, =, \geq\}$
- **Axioms:** 17 axioms (see Manna's book)

> The inclusion of multiplication allows a formula like $\exists x.\ x^2 = 3$ to be expressed (x^2 abbreviates $x \times x$). This formula should be $\mathcal{T}_{\mathbb{R}}$-valid, since the assignment $x \mapsto \sqrt{3}$ satisfies $x^2 = 3$.

- $\mathcal{T}_{\mathbb{R}}$ is decidable (Tarski, 1949). However, it has a high time complexity (doubly exponential).

Difference arithmetic

- *Difference logic* is a fragment (a sub-theory) of linear arithmetic.
- Atomic formulas have the form $x - y \leq c$, for variables x and y and constant c.
- Conjunctions of difference arithmetic inequalities can be checked very efficiently for satisfiability by searching for negative cycles in weighted directed graphs.

  Graph representation: each variable corresponds to a node, and an inequality of the form $x - y \leq c$ corresponds to an edge from y to x with weight c.
- The quantifier-free satisfiability problem is solvable in $\mathcal{O}(|V||E|)$.

Arrays $\mathcal{T}_{\mathsf{A}}$ and $\mathcal{T}_{\mathsf{A}}^{=}$

- Arrays are modeled in logic as applicative data structures.
- **Vocabulary:** $\mathcal{V}_{\mathsf{A}} = \{read, write, =\}$
- **Axioms:**
  - (reflexivity), (symmetry) and (transitivity) of $\mathcal{T}_{\mathsf{E}}$
  - $\forall a, i, j.\ i = j \rightarrow read(a, i) = read(a, j)$
  - $\forall a, i, j, v.\ i = j \rightarrow read(write(a, i, v), j) = v$
  - $\forall a, i, j, v.\ \neg(i = j) \rightarrow read(write(a, i, v), j) = read(a, j)$
- $=$ is only defined for array elements.
- $\mathcal{T}_{\mathsf{A}}^{=}$ is the theory $\mathcal{T}_{\mathsf{A}}$ plus an axiom (extensionality) to capture $=$ on arrays.
  - $\forall a, b.\ (\forall i.\ read(a, i) = read(b, i)) \leftrightarrow a = b$
- Both $\mathcal{T}_{\mathsf{A}}$ and $\mathcal{T}_{\mathsf{A}}^{=}$ are undecidable. But their quantifier-free fragments are decidable.
- Alternative fragments are often preferred that subsume the quantifier-free fragment (allowing restricted forms of index quantification).

Other theories

- *Fixed-size bit-vectors*
  - Model bit-level operations of machine words, including 2^n-modular operations (where n is the word size), shift operations, etc.
  - Decision procedures for the theory of fixed-size bit vectors often rely on appropriate encodings in propositional logic.
- *Algebraic data structures*
  - The theories describe data structures that are ubiquitous in programming like lists, stacks, binary trees, etc.
  - These theories are built around the theory of equality with uninterpreted functions, and are normally efficiently decidable for the quantifier-free fragment.

Combining theories

- In practice, the most of the formulae we want to check need a combination of theories.

Checking $x + 2 = y \to f(read(write(a, x, 3), y - 2)) = f(y - x + 1)$
involves 3 theories: equality and uninterpreted functions, arrays and arithmetic.

- Given theories $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $\mathcal{V}_1 \cap \mathcal{V}_2 = \{=\}$, the *combined theory* $\mathcal{T}_1 \cup \mathcal{T}_2$ has vocabulary $\mathcal{V}_1 \cup \mathcal{V}_2$ and axioms $A_1 \cup A_2$

  [Nelson&Oppen, 1979] showed that if

  - satisfiability of the quantifier-free fragment of $\mathcal{T}_1$ is decidable,
  - satisfiability of the quantifier-free fragment of $\mathcal{T}_2$ is decidable, and
  - certain technical requirements are met,

  then the satisfiability in the quantifier-free fragment of $\mathcal{T}_1 \cup \mathcal{T}_2$ is decidable.

- Most methods available are based on the Nelson-Oppen combination method.

SMT solvers

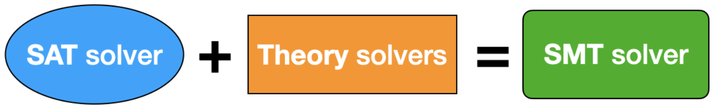
Satisfiability Modulo Theories

- The *Satisfiability Modulo Theories (SMT) problem* is a variation of the SAT problem for first-order logic, with the interpretation of symbols constrained by (a combination of) specific theories (i.e., it is the problem of determining, for a theory $\mathcal{T}$ and given a formula ϕ, whether ϕ is $\mathcal{T}$-satisfiable).

- **SMT solvers** address this problem. They are the core engine of many tools for program analysis, program verification, test-cases generation, bounded model checking of software, modeling, planning and scheduling, ...

SMT solvers

- **SMT solvers** use as building blocks a propositional **SAT solver**, and state-of-the-art **theory solvers**
  - theories need not be finitely or even first-order axiomatizable
  - specialized inference methods are used for each theory



SAT solver + **Theory solvers** = **SMT solver**

- **Equality + UF**
- **Linear Integer Arithmetic**
- **Reals**
- **Arrays**
- **Bit-vectors**
- **...**

Lifting SAT technology to SMT

For a lot of theories one has (efficient) decision procedures for a limited kind of input problems: **sets (or conjunctions) of literals.**

- To deal with boolean combinations of literals there are two main approaches:
  - **Eager approach**
    - translate into an equisatisfiable propositional formula
    - feed it to any SAT solver
  - **Lazy approach**
    - abstract the input formula to a propositional one
    - feed it to a (DPLL-based) SAT solver
    - use a theory decision procedure to refine the formula and guide the SAT solver

---

SMT solvers

- SMT provers have undergone dramatic progress in efficiency and expressiveness. A key ingredient is the SMT-LIB[1] is an international initiative aimed at facilitating research and development in SMT theories.

SMT-LIB
THE SATISFIABILITY MODULO THEORIES LIBRARY

- Its website is an online resource that proposes, as a standard, a unified notation and provides a collection of benchmarks for performance evaluation and comparison of tools, and many usefull information.
- Some SMT solvers: Z3, CVC4, Alt-Ergo,Yices 2, MathSAT, Boolector, ...
- Usually, SMT solvers accept input either in a proprietary format or in SMT-LIB format.

---

[1] http://smtlib.cs.uiowa.edu

The SMT-LIB repository

- Catalog of theory declarations - semi-formal specification of theories of interest
  - A theory defines a vocabulary of sorts and functions. The meaning of the theory symbols are specified in the theory declaration.
- Catalog of logic declarations - semi-formal specification of fragments of (combinations of) theories
  - A logic consists of one or more theories, together with some restrictions on the kinds of expressions that may be used within that logic.
- Check the website http://smtlib.cs.uiowa.edu

The SMT-LIB language

- Textual, command-based I/O format for SMT solvers.
  - Two versions: SMT-LIB 1, SMT-LIB 2 (last version: 2.6)
  - Intended mostly for machine processing.

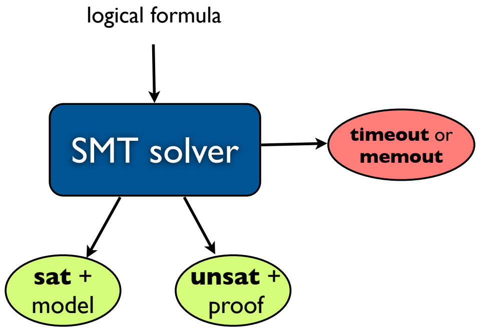
All input to and output from a conforming solver is a sequence of one or more *S-expressions*

$$\langle \text{S-exp} \rangle \ ::= \ \langle \text{token} \rangle \mid (\langle \text{S-exp} \rangle^{*})$$

- SMT-LIB language expresses logical statements in a many-sorted first-order logic. Each well-formed expression has a unique *sort* (type).
- Typical usage:
  - Asserting a series of logical statements, in the context of a given logic.
  - Checking their satisfiability in the logic.
  - Exploring resulting models (if SAT) or proofs (if UNSAT)

Theorem provers / SAT checkers

ϕ is valid iff $\neg\phi$ is unsatisfiable



It may happen that, for a given formula, a SMT solver returns a timeout, while another SMT solver returns a concrete answer.

SMT-LIB 2 example

```
(set-logic QF_UFLIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (distinct x y z))
(assert (> (+ x y) (* 2 z)))
(assert (>= x 0))
(assert (>= y 0))
(assert (>= z 0))
(check-sat)
(get-model)
(get-value (x y z))
```

```
sat
(model  (define-fun z () Int 1)
        (define-fun y () Int 0)
        (define-fun x () Int 3)  )
( (x 3) (y 0) (z 1) )
```

SMT-LIB 2 example

```
(set-logic QF_UFLIA)
(set-option :produce-unsat-cores true)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (! (distinct x y z) :named a1))
(assert (! (> (+ x y) (* 2 z)) :named a2))
(assert (! (>= x 0) :named a3))
(assert (! (>= y 0) :named a4))
(assert (! (>= z 0) :named a5))
(assert (! (>= z x) :named a6))
(assert (! (> x y) :named a7))
(assert (! (> y z) :named a8))
(check-sat)
(get-unsat-core)
```

```
unsat
(a7 a2 a6)
```

SMT-LIB 2 example

Logical encoding of the C program:

```
x = x + 1;
a[i] = x + 2;
y = a[i];
```

- We use the logic QF_AUFLIA (*quantifier-free linear formulas over the theory of integer arrays extended with free sort and function symbol*).
- An access to array `a[i]` is encoded by `(select a i)`.
- An assigment `a[i] = v` is encoded by `(store a i v)`. The result is a new array in everything equal to array `a` except in position `i` which now has the value `v`.
- Assignments such as `x = x+1` are encoded by introducing variables (e.g. `x0` and `x1`) which represent the value of `x` before and after the assignment. The logical encoding would be in this case `(= x1 (+ x0 1))`.

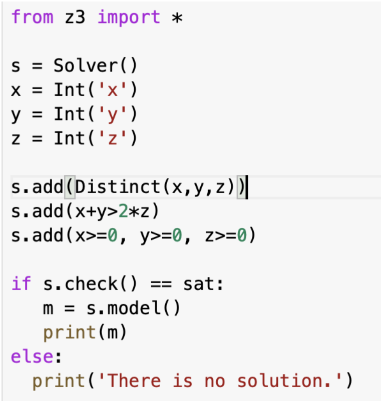
SMT-LIB 2 example

```
(set-logic QF_AUFLIA)
;; Logical encoding of the C program:
;;                                      x = x + 1;
;;                                      a[i] = x + 2;
;;                                      y = a[i];
(declare-const a0 (Array Int Int))
(declare-const a1 (Array Int Int))
(declare-const i0 Int)
(declare-const x0 Int)
(declare-const x1 Int)
(declare-const y1 Int)

(assert (= x1 (+ x0 1)))
(assert (= a1 (store a0 i0 (+ x1 2))))
(assert (= y1 (select a1 i0)))
;; Is it true that after the execution of the program y>x holds?
(assert (not (> y1 x1)))
(check-sat)                          ;; Yes!
```

```
unsat
```

SMT solvers APIs

- Several SAT solvers have APIs for different programming languages that allow an incremental use of the solver.
- For instance, Z3Py: the Z3 Python API.

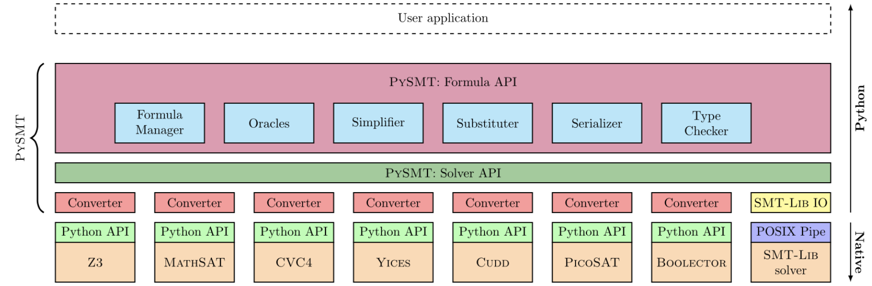


```
from z3 import *

s = Solver()
x = Int('x')
y = Int('y')
z = Int('z')

s.add(Distinct(x,y,z))
s.add(x+y>2*z)
s.add(x>=0, y>=0, z>=0)

if s.check() == sat:
   m = s.model()
   print(m)
else:
  print('There is no solution.')
```

---

pySMT library

- The pySMT[2] library allows a Python program to communicate with several SMT solvers based on a common language.
- This makes it possible to code a problem independently of the SMT solver, and run the same problem with several SMT solvers.



User application

PySMT: Formula API

Formula Manager, Oracles, Simplifier, Substituter, Serializer, Type Checker

PySMT: Solver API

Converter	Converter	Converter	Converter	Converter	Converter	Converter	SMT-Lib IO
Python API	Python API	Python API	Python API	Python API	Python API	Python API	POSIX Pipe
Z3	MathSAT	CVC4	Yices	CUDD	PicoSAT	Boolector	SMT-Lib solver

---

[2] www.pysmt.org

Z3Py Sudoku example

See the Colab notebook with the Sudoku puzzle using the Z3 API for Python.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

SW bug catching using SAT/SMT technology

Main ideia:
Given a program and a claim use a SAT/SMT solver to find whether there exists an execution that violates the claim.

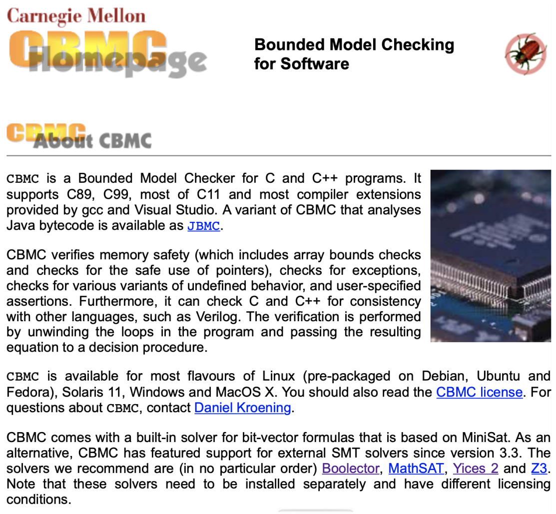
How does it work?

- Transform a program into a set of equations.
  1. Simplify control flow.
  2. Unwind all the loops.
  3. Convert into Single Assignment form.
  4. Convert into equations.
- Solve the equations with a SAT/SMT solver.
- Convert the SAT assignment (if any) into a counterexample.

SW bug catching using SAT/SMT technology

- **Bounded Model Checking of SW** explores this ideia.
- A successful example is the tool **CBMC**: a bounded model checker for C programs.
  - CBMC demonstrates the violation of assertions in C programs, or proves safety of the assertions under a given bound.
  - CBMC implements a bit-precise translation of an input C program, annotated with assertions and with loops unrolled to a given depth, into a formula. If the formula is satisfiable, then an execution leading to a violated assertion exists.
  - CBMC is not able to prove correctness for programs with unbounded loops in general, but is very useful for bug catching.

CBMC: a bounded model checker for C and C++ programs



Bounded Model Checking for Software

About CBMC

CBMC is a Bounded Model Checker for C and C++ programs. It supports C89, C99, most of C11 and most compiler extensions provided by gcc and Visual Studio. A variant of CBMC that analyses Java bytecode is available as JBMC.

CBMC verifies memory safety (which includes array bounds checks and checks for the safe use of pointers), checks for exceptions, checks for various variants of undefined behavior, and user-specified assertions. Furthermore, it can check C and C++ for consistency with other languages, such as Verilog. The verification is performed by unwinding the loops in the program and passing the resulting equation to a decision procedure.

CBMC is available for most flavours of Linux (pre-packaged on Debian, Ubuntu and Fedora), Solaris 11, Windows and MacOS X. You should also read the CBMC license. For questions about CBMC, contact Daniel Kroening.

CBMC comes with a built-in solver for bit-vector formulas that is based on MiniSat. As an alternative, CBMC has featured support for external SMT solvers since version 3.3. The solvers we recommend are (in no particular order) Boolector, MathSAT, Yices 2 and Z3. Note that these solvers need to be installed separately and have different licensing conditions.

Logical enconding of a branching program

The logical enconding of a branching program should be performed as follows:

1. Convert the program into single-assignment (SA) form.
2. Convert the SA program into conditional normal form (CNF).
3. Convert each CNF statement into a logical formula.

Let us see each of these steps...

Single-assignment (SA) form

- A program is in *single-assignment (SA) form* if all its variables satisfies the following proprety:
  - *Once a variable has been used (i.e., read or assigned) it is not assigned again.*
- To encode a (branching) program into a logical formula, one should first convert the program into a SA form in wich multiple indexed version of each variable are used (a new version for each assignment made).

original program
$x = y + z;$
if $(x > y)$
    $x = x + 10$
else
    $y = x + 5;$
$z = x + y;$

$\Longrightarrow$

single-assignment form
$x_1 = y_0 + z_0;$
if $(x_1 > y_0)$
    $x_2 = x_1 + 10;$
else
    $y_1 = x_1 + 5;$
$x_3 = x_1 > y_0 \ ? \ x_2 : x_1;$
$y_2 = x_1 > y_0 \ ? \ y_0 : y_1;$
$z_1 = x_3 + y_2;$

Conditional normal form

The second step for the logical enconding of the program is to convert the SA program into *conditional normal form*: a sequence of statements of the form `(if b then S)`, where `S` is an atomic statement.

- The idea is that every atomic statement is guarded by the conjunction of the conditions in the execution path leading to it.

single-assignment form
$x_1 = y_0 + z_0;$
if $(x_1 > y_0)$
    $x_2 = x_1 + 10;$
else
    $y_1 = x_1 + 5;$
$x_3 = x_1 > y_0 \ ? \ x_2 : x_1;$
$y_2 = x_1 > y_0 \ ? \ y_0 : y_1;$
$z_1 = x_3 + y_2;$

$\Longrightarrow$

conditional normal form
$x_1 = y_0 + z_0;$
if $(x_1 > y_0)$ $x_2 = x_1 + 10;$
if $(!(x_1 > y_0))$ $y_1 = x_1 + 5;$
$x_3 = x_1 > y_0 \ ? \ x_2 : x_1;$
$y_2 = x_1 > y_0 \ ? \ y_0 : y_1;$
$z_1 = x_3 + y_2;$

Program model in SMT-LIB 2

```
(set-logic QF_LIA)
(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(declare-fun y0 () Int)
(declare-fun y1 () Int)
(declare-fun y2 () Int)
(declare-fun z0 () Int)
(declare-fun z1 () Int)

(assert (= x1 (+ y0 z0)))
(assert (=> (> x1 y0) (= x2 (+ x1 10))))
(assert (=> (not (> x1 y0)) (= y1 (+ x1 5))))
(assert (= x3 (ite (> x1 y0) x2 x1)))
(assert (= y2 (ite (> x1 y0) y0 y1)))
(assert (= z1 (+ x3 y2)))
```

Checking properties of the program

original program
$x = y + z;$
if $(x > y)$ {
    $x = x + 10;$
    assert $z > 0;$
} else
    $y = x + 5;$
$z = x + y;$
assert $x == 0$

$\Longrightarrow$

single-assignment form
$x_1 = y_0 + z_0;$
if $(x_1 > y_0)$ {
    $x_2 = x_1 + 10;$
    assert $z_0 > 0;$
} else
    $y_1 = x_1 + 5;$
$x_3 = x_1 > y_0 \ ? \ x_2 : x_1;$
$y_2 = x_1 > y_0 \ ? \ y_0 : y_1;$
$z_1 = x_3 + y_2;$
assert $x_3 == 0;$

Checking properties of the program

single-assignment form

```
$x_1 = y_0 + z_0$;
if $(x_1 > y_0)$ {
    $x_2 = x_1 + 10$;
    assert $z_0 > 0$;
} else
    $y_1 = x_1 + 5$;
$x_3 = x_1 > y_0\ ?\ x_2 : x_1$;
$y_2 = x_1 > y_0\ ?\ y_0 : y_1$;
$z_1 = x_3 + y_2$;
assert $x_3 == 0$;
```

$\Longrightarrow$

conditional normal form

```
$x_1 = y_0 + z_0$;
if $(x_1 > y_0)$  $x_2 = x_1 + 10$;
if $(x_1 > y_0)$  assert $z_0 > 0$;
if $(!(x_1 > y_0))$  $y_1 = x_1 + 5$;
$x_3 = x_1 > y_0\ ?\ x_2 : x_1$;
$y_2 = x_1 > y_0\ ?\ y_0 : y_1$;
$z_1 = x_3 + y_2$;
assert $x_3 == 0$;
```

Checking properties of the program

The properties to be checked are: $(x_1 > y_0 \to z_0 > 0)$ and $(x_3 = 0)$.

Let Γ be the set of formulas of the logical encoding of the program and F be the property to be checked.

$\Gamma \models_{\mathcal{T}} F$ iff no computation path of the program violates the property.

- If $\Gamma \wedge \neg F$ is satisfiable, a counter-example can be shown.
- The $\mathcal{T}$-models of $\Gamma \wedge \neg F$ (if any) correspond to the execution paths of the program that lead to the assertion violation.

Enconding in SMT-LIB 2

```
(set-logic QF_LIA)
...
(assert (= x1 (+ y0 z0)))
(assert (=> (> x1 y0) (= x2 (+ x1 10))))
(assert (=> (not (> x1 y0)) (= y1 (+ x1 5))))
(assert (= x3 (ite (> x1 y0) x2 x1)))
(assert (= y2 (ite (> x1 y0) y0 y1)))
(assert (= z1 (+ x3 y2)))
(push)
(assert (not (=> (> x1 y0) (> z0 0))))
(check-sat)
(pop)
(push)
(assert (not (= x3 0)))
(check-sat)
(get-model)
```

Enconding in SMT-LIB 2

```
unsat
sat
(
(define-fun y2 () Int
4)
(define-fun z0 () Int
0)
(define-fun x2 () Int
10)
(define-fun y1 () Int
4)
(define-fun x1 () Int
(- 1))
(define-fun x3 () Int
(- 1))
(define-fun y0 () Int
(- 1))
(define-fun z1 () Int
3)
)
```

Therefore $(x_1 > y_0 \to z_0 > 0)$ **is valid.**

Therefore $(x_3 = 0)$ **doen't hold.**

`Here is a conter-example:`

Z3Py enconding

See the Colab notebook with the logical enconding of the program.

An example with nested ifs

Here is an example of the transformation of a program with nested if-commands, towards its logical enconding.

original program

```
if ($x > 10$)
{
    $x = x - 10$;
    if ($y < 0$)
        $x = x - y$;
    else
        $x = x + y$;
}
$r = x + x$;
```

$\Longrightarrow$

single-assignment form

```
if ($x_0 > 10$)
{
    $x_1 = x_0 - 10$;
    if ($y_0 < 0$)
        $x_2 = x_1 - y_0$;
    else
        $x_3 = x_1 + y_0$;
    $x_4 = y_0 < 0$ ? $x_2 : x_3$;
}
$x_5 = x_0 > 10$ ? $x_4 : x_0$;
$r_1 = x_5 + x_5$;
```

An example with nested ifs

single-assignment form

```
if ($x_0 > 10$)
{
    $x_1 = x_0 - 10$;
    if ($y_0 < 0$)
        $x_2 = x_1 - y_0$;
    else
        $x_3 = x_1 + y_0$;
    $x_4 = y_0 < 0$ ? $x_2 : x_3$;
}
$x_5 = x_0 > 10$ ? $x_4 : x_0$;
$r_1 = x_5 + x_5$;
```

$\Longrightarrow$

conditional normal form

```
if ($x_0 > 10$)  $x_1 = x_0 - 10$;
if ($x_0 > 10$ && $y_0 < 0$)  $x_2 = x_1 - y_0$;
if ($x_0 > 10$ && !($y_0 < 0$))  $x_3 = x_1 + y_0$;
if ($x_0 > 10$)  $x_4 = (y_0 < 0)$ ? $x_2 : x_3$;
$x_5 = x_0 > 10$ ? $x_4 : x_0$;
$r_1 = x_5 + x_5$;
```

An example with nested ifs

conditional normal form

```
if ($x_0 > 10$)  $x_1 = x_0 - 10$;
if ($x_0 > 10$ && $y_0 < 0$)  $x_2 = x_1 - y_0$;
if ($x_0 > 10$ && !($y_0 < 0$))  $x_3 = x_1 + y_0$;
if ($x_0 > 10$)  $x_4 = (y_0 < 0)$ ? $x_2 : x_3$;
$x_5 = x_0 > 10$ ? $x_4 : x_0$;
$r_1 = x_5 + x_5$;
```

$\Longrightarrow$

logical constraints

$$
\begin{aligned}
& x_0 > 10 \to x_1 = x_0 - 10 \\
& x_0 > 10 \land y_0 < 0 \to x_2 = x_1 - y_0 \\
& x_0 > 10 \land \neg(y_0 < 0) \to x_3 = x_1 + y_0 \\
& x_0 > 10 \land y_0 < 0 \to x_4 = x_2 \\
& x_0 > 10 \land \neg(y_0 < 0) \to x_4 = x_3 \\
& x_0 > 10 \to x_5 = x_4 \\
& \neg(x_0 > 10) \to x_5 = x_0 \\
& r_1 = x_5 + x_5
\end{aligned}
$$