# Laboratory: *SMT Solving and Quantifier Elimination*

## Objectives

1. Applications of SMT solving and quantifier elimination over reals.

## 1 SMT Solving

### 1.1 Problem 1: Scheduling

We consider the following *scheduling* problem [DMB11]. Consider the job-shop-scheduling problem, involving $n$ jobs, each composed of $m$ tasks of varying duration that must be performed consecutively on $m$ machines. The start of a new task can be delayed as long as needed in order for a machine to become available, but tasks cannot be interrupted once they are started. The problem involves essentially two types of constraints:

1. *Precedence.* Between two tasks in the same job
2. *Resource.* Specifying that no two different tasks requiring the same machine are able to execute at the same time.

Given a total maximum time $max$ and the duration of each task, the problem consists of deciding whether there is a schedule such that the end-time of every task is less than or equal to $max$ time units. We use $d_{i,j}$ to denote the duration of the $j$-th task of job $i$. A schedule is specified by the start-time $t_{i,j}$ for the $j$-th task of every job $i$. The job-shop-scheduling problem can be encoded in SMT using the theory of linear arithmetic. A precedence constraint between two consecutive tasks $t_{i,j}$ and $t_{i,j+1}$is encoded using the inequality $t_{i,j+1} \geq t_{i,j} + d_{i,j}$; this inequality states that the start-time of task $j+1$ must be greater than or equal to the start time of task $j$ plus its duration. A resource constraint between two tasks from different jobs $i$ and $i'$ requiring the same machine $j$ is encoded using the formula $(t_{i,j} \geq t_{i',j} + d_{i',j}) \vee (t_{i',j} \geq t_{i,j} + d_{i,j})$, stating the two tasks do not overlap. The start time of the first task of every job $i$ must be greater than or equal to zero, so the result is $t_{i,1} \geq 0$. Finally, the end time of the last task must be less than or equal to $max$, hence $t_{i,m} + d_{i,m} \leq max$.

An instance of the problem is $max = 8$ as well as

| $d_{ij}$ | Machine 1 | Machine 2 |
|---|---|---|
| Job 1 | 2 | 1 |
| Job 2 | 3 | 1 |
| Job 3 | 2 | 3 |

The logical formula encoding the scheduling problem combines logical connectives (conjunctions, disjunction, and negation) with atomic formulas in the form of linear arithmetic inequalities. We call it an SMT formula. We can use Z3 SMT solver (`https://rise4fun.com/Z3`) for checking if the formula is SAT or UNSAT, and in the first case a satisfying assignment.

The input of Z3 are constructs in SMT-LIB format `http://smtlib.cs.uiowa.edu`. For the problem above, we have:

```
(set-logic QF_LIA); the underlying logic - quantifier-free linear integer arithmetic
(declare-fun t11 () Int); declare constant t11
...
(assert (>= t11 0)); define that t11 >= 0
...
(check-sat); check if the formula above is SAT
(get-model); provide a model (satisfying assignment) if SAT
(exit)
```

## 1.2 Problem 2: Dynamic Symbolic Execution

Dynamic symbolic execution [Cad16] is a technique for automatically exploring paths through a program:

- Determines the feasibility of each explored path using a constraint solver
- Checks if there are any values that can cause an error on each explored path
- For each path, can generate a concrete input triggering the path

SMT solvers play a central role in dynamic symbolic execution. A number of tools used in industry are based on dynamic symbolic execution (e.g. CUTE, KLEE, DART, SAGE, Pex, Yogi) designed to collect explored program paths as formulas, using solvers to identify new test inputs that can steer execution into new branches. SMT solvers are suitable for symbolic execution because the semantics of most program statements are easily modeled using theories supported by these solvers. To illustrate the basic idea of dynamic symbolic execution, consider the greatest common divisor algorithm below, taking the inputs $x$ and $y$ and producing the greatest common divisor of them.

```
int GCD (int x, int y){
     while (true) {
          int m = x % y;
          if (m == 0) return y;
          x = y;
          y = m;
     }
}
```

The following piece of code represents the unfolding of the loop twice.

```
int GCD (int x_0, int y_0){
     int m_0 = x_0 % y_0;
     assert (m_0 != 0)
     int x_1 = y_0;
     int y_1 = m_0;
     int m_1 = x_1 % y_1;
     assert (m_1 = 0)
}
```

Assertions are used to enforce that the condition of the `if` statement is not satisfied in the first iteration and is in the second iteration. The sequence of instructions is equivalently represented as a formula where the assignment statements have been turned into equations. The resulting path formula is satisfiable (check it via Z3!). The satisfying assignment can be used as input to the original program causing the loop to be entered twice, as expected.

## 2 Quantifier Elimination

A computer algebra system (CAS), such as Mathematica (`https://www.wolfram.com/mathematica/`), Maple (`https://www.maplesoft.com/products/Maple/`), Sage (`http://www.sagemath.org/`), is a mathematical software with the ability to manipulate mathematical expressions in a way similar to the traditional manual computations of mathematicians and scientists. We will use Mathematica 6.0 (available installation in the lab) for exemplifying quantifier elimination.

- universal quantifier: `ForAll[x, expr]`, `ForAll[x1,...,xn, expr]`, `ForAll[x, cond, expr]`, `ForAll[x1,...,xn, cond, expr]`.
- existential quantifier: `Exists[x, expr]`, `Exists[x1,...,xn, expr]`, `Exists[x, cond, expr]`, `Exists[x1,...,xn, cond, expr]`.
- command implementing quantifier elimination: `Resolve[expr]`, `Resolve[expr, dom]`, `Reduce[expr, vars]`, `Reduce[expr, vars, dom]` (`dom` can be `Reals`, `Integers`, `Complexes`).

See examples in `QuantifierElimination.nb` file.

### 2.1 Exercise

Consider the following algorithm:

```
L = min(1,x)
U = max(1,x)
while abs(U-L) > eps
    L = L + (x-L^2)/(L+U)
    U = U + (x-U^2)/2U
return U-L
```

The precondition of the algorithm is $x\in\mathbb{R}\wedge x>0$ and the postcondition is $U - L<eps \wedge L\leq y\leq U \wedge y^2=x$. Come up with a loop invariant and use it to prove automatically the correctness of the algorithm. In the proof, since the verification conditions are quantified formulae over reals, you might want to use Mathematica (`Reduce` command) for the proof. (Fill out the `nb` file `QuantifierElimination.nb`).

## References

[Cad16] Cristian Cadar. An introduction to dynamic symbolic execution and the KLEE infrastructure, 2016. Invited talk @ International Symposium on Software Testing and Analysis (ISSTA 2016).

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. Commun. ACM, 54(9):69–77, September 2011.