VRS — Summer 2014	Problem Set #2
Computer-Aided Verification	Due June 18, 2014

There are two kinds of problems: problems marked "T" are theoretical problems, and problems marked "E" involve coding and experimentation.

Problem T1.1. (a) Give a procedure to count the number of satisfying assignments of a Boolean formula represented as a BDD that runs in time linear in the size of the BDD. (b) Let $\varphi(x, y)$ be a Boolean formula. The *universal quantification* of $\varphi(x, y)$ is defined as

$$\forall x.\varphi(x, y) \equiv \varphi(0, y) \wedge \varphi(1, y)$$

Show how you can implement universal quantification using BDD operations.

Problem T1.2. How will you extend the symbolic execution engine discussed in the class if we have pointers to variables and an allocation function that allocates an integer and returns a pointer to the allocated memory? That is, in addition to integer variables (as in the class), suppose your program is allowed to have pointer variables and the following operations on them:

$$p := \textbf{nul} \;//\; \text{assign pointer variable to null} \tag{1}$$
$$p := \textbf{new}() \;//\; \text{allocates a block of memory to store an integer and stores the address in } p \tag{2}$$
$$p := q \;//\; \text{assign pointer variable } q \text{ to pointer variable } p \tag{3}$$
$$*p := e \;//\; \text{assign the integer expression } e \text{ to the memory pointed to by } p \tag{4}$$

and the conditional operations

$$p = \textbf{nul} \qquad p \neq \textbf{nul} \tag{5}$$
$$p = q \qquad p! = q \tag{6}$$

Sketch (on paper) how you will maintain the symbolic store and the path constraint in the presence of these operations. (a) Will you be able to check that the program never dereferences a null pointer along a path? (b) Will your implementation change if we add an explicit **free** operation to free memory? Will you be able to check that a program does not reference memory that has been freed?

Problem T1.3. Cartesian predicate abstraction. The predicate abstraction algorithm presented in class runs in time exponential in the number of predicates. Here is an algorithm that computes an approximation to the predicate abstraction but makes a linear number of SMT calls: Given R and predicates $p_1, \ldots, p_n$, define the variables $x_1, \ldots, x_n$ as: $x_i = p_i$ if $R \Rightarrow p_i$, $x_i = \neg p_i$ if $R \Rightarrow \neg p_i$, and $x_i = \mathit{true}$ otherwise, for each $i \in \{1, \ldots, n\}$. Define the Cartesian abstraction $CA(R, \{p_1, \ldots, p_n\})$ of R w.r.t. the set $\{p_1, \ldots, p_n\}$ as the conjunction

$$\bigwedge_{i=1}^{n} x_i$$

The algorithm clearly makes a linear number of calls to the SMT solver.

Show that

$$PA(R, \{p_1, \ldots, p_n\}) \Rightarrow CA(R, \{p_1, \ldots, p_n\})$$

The Cartesian abstraction gives a "coarser" abstraction than predicate abstraction but is quicker to compute. Many software model checkers implement Cartesian abstraction instead of predicate abstraction for scalability, and in many (but by no means, all) examples, Cartesian abstraction is enough to prove or disprove a property.

As an example, suppose an abstraction is done in terms of the predicates p_1, p_2, p_3 and p_4, and $R = ((p_1 \wedge \neg p_2) \vee (p_1 \wedge p_3)) \wedge \neg p_4$. Then

$$CA(R, \{p_1, \ldots, p_4\}) = p_1 \wedge \mathit{true} \wedge \mathit{true} \wedge \neg p_4 \,.$$

Problem E1.4. This assignment will give you some experience using SMT solvers. Download the Z3 SMT solver from Microsoft Research (`https://z3.codeplex.com/`). Depending on the language of your choice, here are some pointers on how to use it:

C/C++ Look at the examples that come with Z3.

Java You will need the unstable branch of Z3, then look at the examples that come with Z3.

Python Again, look at the examples that come with Z3. Additionally, there used to be a nice tutorial at `http://rise4fun.com/z3py/tutorial`. If it does not become available any time soon, you can get the raw version of the tutorial here: `http://goo.gl/HOooly`.

Ocaml See this discussion: `https://z3.codeplex.com/discussions/473552`.

Haskell Use the sbv package: `http://hackage.haskell.org/package/sbv`.

Scala Use ScalaZ3: `https://github.com/epfl-lara/ScalaZ3`.

In this problem, your task is to implement a simple method for Petri net verification. Start by reading sections 2 and 3 from the following paper: `http://www.mpi-sws.org/~fniksic/files/cav2014.pdf`. The method in question is explained in Section 3 of that paper.

Your implementation should input a Petri net reachability problem instance in MIST format, which is explained here: `https://github.com/pierreganty/mist/wiki`. For your convenience, we have provided a Python library for parsing this format: `https://github.com/fniksic/mister`.

You can find a lot of examples for testing your implementation here: `http://www.mpi-sws.org/~fniksic/files/examples.tar.gz`. Examples are contained in the `*.spec` files. In those examples, the set of target states represents the set of error states. Therefore, if a Petri net can reach it, it is unsafe. For how many examples can your implementation prove safety? For how many safe examples is the method inconclusive? (To know which examples are safe, check the `*.meta` files.)

Problem E1.5 (bonus). We will consider the problem of checking that a compiler transformation has not introduced bugs in a program. That is, we are given two versions of a program, the original one (P) and one that a compiler produces after one optimization step (P'), and we would

like to check that they produce the same results for every input. For simplicity, assume that the programs do not have loops, and are given by the following grammar of operations:

$$
\begin{array}{lrcl}
\text{Expressions} & e & ::= & c \mid x \mid c \cdot x \mid e + e \mid e - e \\
\text{Predicates} & b & ::= & e = e \mid e \neq e \mid e >= e \mid e <= e \mid e > e \mid e < e \\
\text{Statements} & s & ::= & \mathsf{skip} \mid x := e \mid \mathsf{if}(b)\mathsf{then}\ s\ \mathsf{else}\ s \mid s; s \\
\text{Program} & p & ::= & \mathsf{input}(x_1, \ldots, x_n); s; \mathsf{output}(y_1, \ldots, y_m)
\end{array}
$$

Here, c stands for an integer constant, x for a variable. The statements have the usual meaning.

Write a procedure that takes two programs with the same set of inputs and outputs and checks that they are equivalent by querying the SMT engine. First, formulate the problem as a satisfiability question. Then, implement your procedure.

[Paper Reading] Read the following paper on concolic execution:
K. Sen, D. Marinov, G. Agha. CUTE: A concolic unit testing engine for C, FSE 2005.
(A Google search should find it.) Write a one-paragraph summary for the paper, pointing out the main idea and at least one direction of future work not explicitly mentioned in the paper.

[Paper Reading] Read Bryant's paper on BDDs on the course page.