EECS 598-002 WN 2023
Software Tools Quick Reference Guide

Karem A. Sakallah

March 6, 2023

1 RePyCUDD (`https://github.com/pysmt/repycudd`)

RePyCUDD is a re-entrant Python Wrapper for CUDD. To use it, the Python file should be structured as follows:

1.1 Importing Wrapper Files

```python
# Import the necessary wrappers
import repycudd # Python wrapper for CUDD
import sym_dot  # Python utility to map variable numbers to their names
```

1.2 Creating a BDD "manager"

```python
# Access to BDD operations must be through a so-called BDD manager.
# The following instruction creates a manager called mgr
mgr = repycudd.DdManager()
```

1.3 Creating Variables

```python
# Variable names can be arbitrary strings that start with a letter.
# There are two ways to create variables:
# a) using the IthVar method to assign a specific integer index to the variable name
# b) using the NewVar method to automatically assign the largest integer index to the variable name
x0 = mgr.IthVar(0)
x1 = mgr.IthVar(1)
x2 = mgr.NewVar()               # Gets assigned index 2
```

1.4 Creating Functions

```python
# The following logical operators are available to create and manipulate functions.
# Note that they must all be lower-case except for their first letter which must upper case.
# Arity = 1: Not
# Arity = 2: And, Or, Nand, Nor, Xor, Xnor
# Arity = 3: Ite
# The sample functions here are f = (x0 | (~x1 & x2)) and g = (x1 xor x2) & x0
# For complex expressions, it is better to build the functions incrementally for better
# readability. This is uncessary for these sample functions, but is presented here
# (for function g) for illustration purposes
f = mgr.Or(x0, mgr.And(mgr.Not(x1), x2))
g = mgr.Xor(x1, x2)
g = mgr.And(g, x0)
```

1.5 Generating and Printing a Multi-rooted BDD

```
# To generate the BDD graph:
# a) create a function array,
# b) push the required functions into it,
# c) dump it into a .dot file, and
# d) generate the graph with the UNIX command "dot -Tpdf filename.dot > filename.pdf".
farray = repycudd.DdArray(mgr,2)
farray.Push(mgr.BddToAdd(f))
farray.Push(mgr.BddToAdd(g))
mgr.DumpDotArray(farray, "example.dot")
```

1.6 Using Variable and Function Names in Generated BDD Graph

```
# CUDD labels the BDD graph with variable numbers instead of names.
# It also labels the functions F0, F1, etc., based on the order in which
# they were pushed into the function array. You can override these defaults
# by specifying your own names for both variables and functions and by replacing
# the DumpDotArray method with the "sym_dot_manager" method
# from the sym_dot Python wrapper.
var_names = ["x0", "x1", "x2"]
fcn_names = ["f", "g"]
sym_dot.sym_dot_manager("example.dot", var_names, fcn_names).add_syms()
```

1.7 Abstraction Operations

1.7.1 ExistAbstract: $g(x_m, \cdots, x_1) = \exists y_n, \cdots, y_1 : f(x_m, \cdots, x_1, y_n, \cdots, y_1)$

```
Y = mgr.And(yn, ..., y1)                # Using a composition of 2-input Ands
g = mgr.ExistAbstract(f, Y)
```

1.7.2 AndAbstract: $h(x_m, \cdots, x_1) = \exists y_n, \cdots, y_1 : f(x_m, \cdots, x_1) \wedge g(x_m, \cdots, x_1, y_n, \cdots, y_1)$

```
Y = mgr.And(yn, ..., y1)                # Using a composition of 2-input Ands
h = mgr.AndAbstract(f, g, Y)
```

1.8 Swapping Variables: $f(x_n, \cdots, x_1) \rightarrow f(y_n, \cdots, y_1)$

```
# Create variable arrays
X = repycudd.DdArray(mgr, n)
Y = repycudd.DdArray(mgr, n)
X.Push(xn)
...
X.Push(x1)
Y.Push(yn)
...
Y.Push(y1)
f = mgr.SwapVariables(f, X, Y, n)       # Replace X with Y in f
```

1.9 BDD Statistics

```
mgr.ReadSize()          # Number of BDD variables in existence
mgr.ReadNodeCount()     # Number of live nodes
mgr.ReadReorderings()   # Number of times reorderings
mgr.ReadMemoryInUse()   # Memory usage (bytes)
```

2 PySAT (`https://pysathq.github.io/`)

2.1 Boolean Formula Manipulation (pysat.formula)

The formula module includes classes for manipulating a variety of Boolean formulas. The relevant class for us is CNF which provides methods for maniplulating CNF formulas. In this class

- A **clause** is a comma-separated list of literals (positive and negative integers) enclosed in square brackets, e.g., [-1, 2], and
- A **clause list** is a comma-separated list of clauses enclosed in square brackets, e.g., [[-1, 2], [-2, 3]].

Some of the available methods in the CNF class are:

- f = CNF(from_file='file-name.cnf') reads a CNF formula from a file and assigns it to f.
- f = CNF(from_clauses=clause_list) assigns a clause list to f.
- f.append(clause) appends a single clause to f.
- f.extend(clause list) extends f with a clause list.
- g = f.copy() creates a copy of f.
- f.clauses outputs the clauses of f.
- f.nv outputs the number of variables in f.
- f.negate() creates a CNF for the negation of f using Tseitin auxiliary variables.
- f.to_file('file-name.cnf') writes f to a file.

2.2 SAT solvers' API (pysat.solvers)

The solvers module includes a number of classes for creating and manipulating SAT solvers. The main class in this module is Solver which provides a number of methods including:

- s = Solver(name='solver name') creates a solver s with the given name. Table 1 lists available solvers.
- s = Solver(use_timer=True) records SAT solving time for s.
- s.add_clause(clause) adds a single clause to s.
- s.append_formula(clause_list) adds a clause list to s.
- s.solve()
- s.get_model() outputs a complete satisfying assignment or None.
- s.delete()
- s.nof_clauses() outputs the number of clauses in s.
- s.nof_vars() outputs the number of variables in s.
- s.time() outputs the time of the last SAT call.
- s.time_accum() outputs the accumuated time for all SAT calls.
- s.accum_stats() outputs the low-level accumulated stats (varies per solver)
- s.enum_models() outputs all models of s.

SAT Solver	Short Names
CaDiCaL (rel-1.0.3)	'cd', 'cdl', 'cadical'
Glucose (3.0)	'g3', 'g30', 'glucose3', 'glucose30'
Glucose (4.1)	'g4', 'g41', 'glucose4', 'glucose41'
Lingeling (bbc-9230380-160707)	'lgl', 'lingeling'
MapleLCMDistChronoBT (SAT competition 2018 version)	'mcb', 'chrono', 'maplechrono'
MapleCM (SAT competition 2018 version)	'mcm', 'maplecm'
MapleSat (MapleCOMSPS_LRB)	'mpl', 'maple', 'maplesat'
Minicard (1.2)	'mc', 'mcard', 'minicard'
Minisat (2.2 release)	'm22', 'msat22', 'minisat22'
Minisat (GitHub version)	'mgh', 'msat-gh', 'minisat-gh'

Table 1: Available Solvers

3 AVR (`https://github.com/aman-goel/avr`)

AVR (Abstract Verification of Reachability) is a verifier focused on checking *control-centric* safety properties of hardware designs. It performs *data abstraction* and extends IC3-style incremental induction to the empty theory of first-order logic (FOL), commonly referred to as Equality with Uninterpreted Functions (EUF). Its data abstraction is configurable so that data variables and operations above a use-specified word length are abstracted to EUF and those variables and operations below that word length are interpreted as fixed-width bit vectors.

AVR can verify safety properties of hardware designs written in the Verilog Hardware Description Language[1] (HDL). The property being checked must be specified with an `assert` statement.

The basic usage of AVR is by typing the command

```
$ avr filename.v
```

at the UNIX command prompt.

AVR generates voluminous output consisting of many files organized in a hierarchical directory structure (see Figure 1). The default name of the top directory is, naturally, `output` and can be specified with a full UNIX path prefix. Within this output directory, AVR creates subdirectories that contain the results of specific verification runs. The default run name is `test` and its results are saved in a subdirectory named `work_test`.

The default names for the output directory and its subdirectories can be changed using the command options:

- `-o 'output-dir-name'`
- `-n 'run-name'`

Note that multiple runs with different names can be saved in the same output directory and it's good practice to give the directory as well as the various runs in it unique descriptive names.

The most important files in these directories are:

- `'run-name.results'`: contains various statistics about the design, the number of SAT calls, the number of CTIs, etc.
- 'data/reach.output': contains a detailed trace of the verification run.
- `'inv.txt'` contains the derived inductive invariant in a textual format and `proof.smt2` is a file containing SMT-LIB queries for the safety and inductiveness of the derived invariant. This file can be checked by an SMT solver such as Z3.These files are generated by the `--witness` and `--smt2` command line option.

[1]AVR can also read other formats including BTOR2, a bit-precise format for word-level designs, and VMT-LIB, an extension of the SMT-LIB format for symbolic transition systems.

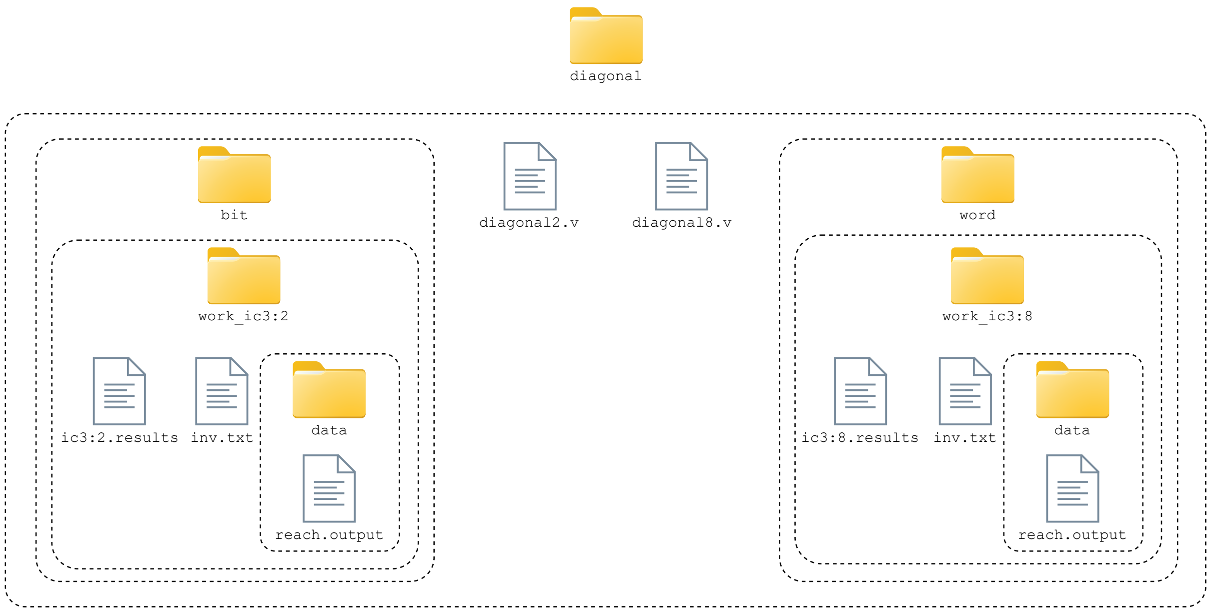

Figure 1: avr's Output Directory Structure

3.1 AVR Verification Options

By default, AVR performs data abstraction on variables and operators whose bit width is greater than 1. This can be modified in a number of ways by the following command line options:

- `--aig`: This option uses the yosys synthesizer to flatten the word-level design to produce an equivalent bit-level implementation. Verification is performed at the bit level with no abstraction.
- `--abstract sa'n'`: this option configures AVR to perform data abstraction for variables and operators whose width is greater than `'n'`. Variables and operators whose width is greater than 1 but less than or equal to `'n'` are encoded as bit vectors of their specified width.
- `--bmc`: this option causes AVR to perform bounded model checking. By default AVR uses a bound of 1000 steps. This bound can be changed by the command line option `--kmax 'n'` for some integer `'n'`. AVR's version of BMC can take advantage of its data abstraction features.

3.2 AVR Output Options

- AVR generates a trace of its execution in `/data/reach.output`. The option `-v n`, where `n` is a number between 0 and 6, determines the verbosity of the output. Higher values of `n` produce more detailed traces.
- `--witness`: Generates inv.txt, a human-readable file containing the inductive invariant (if the property holds) or a counterexample trace (if the property does not hold).
- `--smt2`: Generates proof.smt2, a file in SMT-LIB format containing the queries that establish inductiveness which can be checked by an SMT solver such as Microsoft's Z3.

3.3 Verification using PDR

In addition to causing AVR to perform bit-level verification, the`--aig` option also creates a version of the bit-level design in aig format suitable for use in the PDR verifier.
PDR can be invoked from within the UC Berkeley `abc` synthesis and verification system:

```
$ yosys-abc
abc 01> read file.aig
abc 02> pdr -h
abc 03> pdr
```

Invoking PDR with `-h` lists all available options along with short descriptions of their effect.

4 Z3 (`https://microsoft.github.io/z3guide/`)

Z3 is Microsoft's Satisfiability Modulo Theories (SMT) solver. It can be accessed on CAEN by typing Z3 followed by an smt2 file at the command prompt. The file must contain declarations and assertions in the SMT-LIB format which Z3 can check for satisfiability. Z3 can also be accessed via a web interface at the above URL (which also has extensive documentation). There is also a Python wrapper that I am trying to figure out and will provide more info on later.