A Toolkit for the Theory of Equality

The Tarski Team

Abstract

We introduce Tarski: A toolkit for modeling and verifying systems expressible in the logic of equality. A novel feature of our approach is that we reduce the decision problem to sequential circuit reachability, rather than CNF satisfiability, which paves the way for transformation-based verification as well as the application of semi-formal techniques, simulation and emulation.

Contents

1 Overview

The increasing complexity in the design and verification of complex hardware and software systems has led to modeling and verification of systems at a higher level of abstraction. Higher level abstract models enable designers to work through architectural issues before committing to a low level implementation and also allow verification engineers to verify correct operation at the level at which design was conceived. Tarski is a tool we have developed for modeling and verifying hardware and software systems expressible in the logic of equality. Equality logic is very powerful: diverse verification problems ranging from verification of pipelined processors [6, 4] to checking of correctness of compilers [13] can be expressed in it.

The most common approach to check the satisfiability of equality logic formulas is to create an equi-satisfiable propositional logic formula [9, 3]. Tarski uses the *eager encoding approach*: the *finite domain property* is used to find a small domain for each variable, which is then encoded as a bit-vector. Tarski also enables modeling of systems using equality logic with uninterpreted functions (EUF), where the check is posed as a problem in satisfiability checking for quantifier-free formulas involving uninterpreted functions in addition to the equality operator. Tarski checks the validity of EUF formulas through transformation into equality logic formulas [1].

Rather than using a Boolean Satisfiability (SAT) solver or a BDD package to check the satisfiability of the generated propositional formulas, Tarski compiles the system to synthesizable VHDL. A key feature of Tarski compilation is the ability to synthesize a *sequential executable* model. The sequential executable model of the system not only enables the application of formal verification techniques such as symbolic model-checking [11] to verify the design; simulation, emulation and semi-formal techniques can also be used to validate the design. Sequential modeling paves the way for the application of *transformation-based verification* (TBV) [2] with its ability to leverage various transformations to successively simplify and decompose large problems. We use the SixthSense tool [12] for solving sequential circuits. In addition to the ability to generate sequential executable models, the Tarski toolkit has several other novel features:

1. Models can be specified using C++ or Tcl with methods that make it easy to analyze, simplify and verify designs. The Tcl interface enables rapid prototyping, scripting to enable automation as well as interactive operation of the tool.

2. Tarski has facilities to visualize the design (as a graph using the *dot* package) and enables users to generate useful counterexamples through intelligent naming of signals in the generated VHDL.

3. Through the generation of VHDL, Tarski enables the application of logic synthesis techniques to speed up SAT [7] and thus enhance the verification of satisfiability-preserving combinational VHDL.

Tarski has been implemented in C++, containing around 5000 lines of code. It is highly extensible, and the underlying data structures are documented in detail. Extensive programmer documentation exists for Tarski. The `doxygen` tool has been used to create the documentation.

2 Modeling Designs Using Tarski

Tarski Netlist: A Tarski *netlist* is a directed acyclic graph, where the nodes correspond to *primitive circuit elements*, and the edges correspond to connections between these elements. The primitive circuit elements include *primary inputs, registers, multiplexers, Boolean and scalar constants, uninterpreted functions, uninterpreted relations, equality checkers, 2-input logic gates and inverters*. Some nodes are also labeled as *primary outputs*.

Nodes are of two *types*—Boolean-valued and scalar valued. Primary inputs and registers can either be Boolean or scalar typed. Multiplexers are scalar typed and are required to have a single Boolean-valued input, and two scalar-valued inputs; equality-checkers are Boolean typed and should have two scalar-valued inputs; uninterpreted functions and uninterpreted relations have scalar-valued inputs and are scalar typed and Boolean typed respectively. All 2-input logic gates and inverters are Boolean typed with Boolean-valued inputs.

Registers have two nodes associated with them, the initial-value function and the next-state function. We disallow any registers from appearing in the initial-value functions. Memories and register files in Tarski are modeled as scalar registers, the read and writes to the memory are modeled as uninterpreted functions, with extra constraints are added to relate reads and writes [14]. Our entire verification problem is represented as a Tarski netlist, comprising the *design under verification*, its *environment* and relevant *assertions*.

As an example, consider a trivial processor specified by the following logic:

```
if stall
  then regfile' := regfile;
  else regfile' := write( regfile,
                          dest,
                          alu( op,
                               read( regfile, src )));
```

This system can be specified in C++ using the following code:

```
F
build_spec()
{

  F spec_stall = bi(``spec_stall"); // Scalar input name spec_stall
  F src1  = si("src");  // Scalar input name src
  F dest  = si("dest");  // Scalar input name dest
  F op    = si("op");    // Scalar input name op


  F spec_regfile = sr("mem@specRF"); // Scalar register named specRF
  F init_spec_regfile = si("specinitRF"); // Scalar input named specinitRF

  /*
   * Next state for specRF.  We use several library functions to
   * simplify our life:
```

```
 *
 *   o mux -  takes 3 arguments, the first of which
 *            has to be Boolean, the next two have to be scalar.
 * (This typechecking is performed in the compile step.)
 *
 *   o uif2 - takes 3 arguments, the first is a string that is the
 *     name of the UIF, the other arguments are the inputs to the UIF.
 *
 *   o uif3 - similar to uif2, but takes 3 args
 *
 */


 F next_spec_regfile = mux( spec_stall,
                       spec_regfile,
                       uif3( "mem#write",
                           spec_regfile,
                           dest,
                           uif2("alu",
                               op,
                               uif2("mem@read", spec_regfile, src1 ))));


/*
 * The expression below sets the initial value and next state
 * inputs for the scalar register specRF.
 *
 * Note that whenever we declare a new F node, it's arguments
 * MUST be available.  The only exception is for registers;
 * and this exception is required because if the circuit is
 * sequential, the requirement is impossible.
 */

 Node_AssignRegister(spec_regfile, init_spec_regfile, next_spec_regfile);
 return spec_regfile;
}
```

The function `build_spec` generates internally a parse tree that is depicted pictorially in Figure 1. Models can be specified in either C++ or Tcl in Tarski. The Tcl specification for the processor described above is as follows:

```
# Defining the Specification

# declaration of inputs

set src1 [si "src1"]
set dest [si "dest"]
set op [si "op"]
set spec_stall [bi "spec_stall"]

# specification register file

set spec_regfile [sr "mem@specRF"]
```

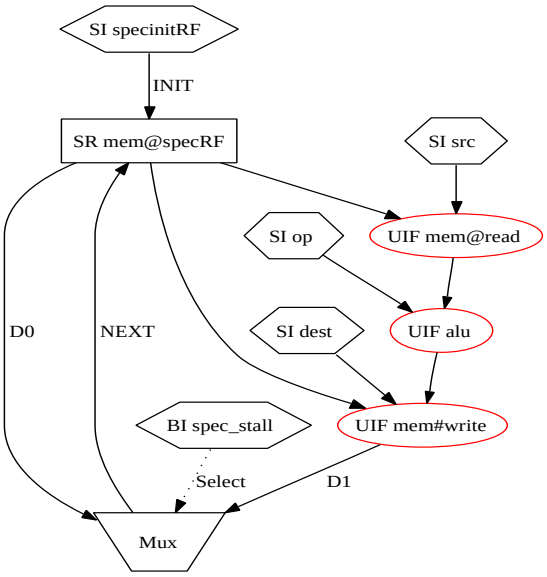

Figure 1: Graphical view of data structure for `spec`.

```
set init_spec_regfile [si "specinitRF"]

# setting up the next state function of regfile

set next_spec_regfile [mux $spec_stall \
                       $spec_regfile \
                       [uif3 "mem\#write" \
                             $spec_regfile \
                             $dest \
                             [uif2 "alu" \
                                   $op \
                                   [uif2 "mem@read" \
                                         $spec_regfile $src1]]]]

# assigning the init and next state functions for regfile

NodeAssignRegister $spec_regfile $init_spec_regfile $next_spec_regfile
```

A 2 stage pipelined implementation of the same processor is given by the following logic:

```
bubble-ex' := stall;
dest-ex'   := dest;
op-ex'     := op;

if ( not bubble-ex ) and ( dest = src )
  then arg1 := result;
  else arg1 := read( regfile, src );

result = alu (op-ex, arg1);
```

```
if bubble-ex
  then regfile' := regfile;
  else regfile' := write( regfile, dest-ex, result );
```

The specification of the 2 stage pipelined implementaion in Tcl is as follows:

```
# Defining the Implementation

# The implementation has two stages decode and execute
# To avoid stalling when the source of the instruction in
# decode is same as the destination of instruction in execute
# we forward the results from execute to decode

# declaration of inputs

set src1 [si "src1"]
set dest [si "dest"]
set op [si "op"]
set impl_stall [bi "impl_stall"]

# implementation register file

set impl_regfile [sr "mem@implRF"]
set init_impl_regfile [si "initRF"]

# Execute Stage pipeline register for stall

set bubble_ex  [br "bubble_ex"]
set init_bubble_ex [bi "init_bubble_ex"]
NodeAssignRegister $bubble_ex $init_bubble_ex $impl_stall

# Execute stage pipeline register for dest

set dest_ex [sr "dest_ex"]
set init_dest_ex [si "init_dest_ex"]
NodeAssignRegister $dest_ex $init_dest_ex $dest


# Execute stage pipeline register for op

set op_ex [sr "op_ex"]
set init_op_ex [si "init_op_ex"]
NodeAssignRegister $op_ex $init_op_ex $op


# Execute stage pipeline register for arg1

set arg1 [sr "arg1"]
set init_arg1 [si "init_arg1"]
```

```
# Result generated in execute stage

set result [uif2 "alu" $op_ex $arg1]

# Next state for arg1

set next_arg1 [mux [[$bubble_ex inv] & [$dest_ex equal $src1]] \
   $result \
   [uif2 "mem@read" $impl_regfile $src1]]

# Assigning arg1 register

NodeAssignRegister $arg1 $init_arg1 $next_arg1

# Setting up the next state function of regfile

set next_impl_regfile [mux $bubble_ex \
   $impl_regfile \
   [uif3 "mem\#write" $impl_regfile $dest_ex $result]]

# Assigning impl_regfile

NodeAssignRegister $impl_regfile $init_impl_regfile $next_impl_regfile
```

2.1 Sequential Modeling

To enable sequential modeling, Tarski needs to identify the register files and memories in the design. Users can enable this identification by properly naming the scalar registers corresponding to register files and memories. Every scalar register that corresponds to a memory/register file needs to have **"mem@"** prefix to the name. The uninterpreted functions correspoing to reading and writing to the register files and memory would need to be associated with the read and write ports of the register files and memory. Users can enable this association by adding a **"mem@"** prefix to the names of the 'read' uninterpreted functions and by adding a **"mem#"** prefix to the names of the 'write' uninterpreted functions.

3 Verification using Tarski

The first step to start verification using Tarski is to invoke the Tarski executable. Tarski is built with a Tcl interface which enables rapid prototyping, scripting to enable automation as well as interactive operation of the tool. The common method of running Tarski is to run it in interactive mode. Invoking the Tarski executable (named **"tarski"**) will open a Tcl shell from which commands can be invoked to build and verify equality logic models.

3.1 Visualizing Designs using Tarski

Tarski has facilities to visualize the design as a graph using the *dot* package. For visualizing the netlist, the user needs to do the following steps,

1. Create a netlist from the formula
2. Print the formula in the dot format
3. Call the Tcl script to generate a postscript/pdf of the generated graph.

A sample Tcl script to visualize the trivial processor depicted pictorially in Figure 1 is given below. The "create_dot.tcl" script names the generated pdf as "foo.pdf".

```
# Visualize the netlist using dot

# we need to first create a Netlist from the formula

set spec_net [Net_CreateFromFormula [$spec_regfile cget -raw]]

# setup a tmp file to print out the Netlist info

set printFile "/tmp/tmp.out"

# call PrintFormula

Net_PrintFormulaWrap $spec_net [$spec_regfile cget -raw] $printFile

# call dot to generate a pdf of the netlist graph

source create_dot.tcl
```

3.2 Synthesis of Tarski model to VHDL

The path from a Tarski model to VHDL is as follows: The memory arrays in the design are identified and distinguished from normal scalar-typed registers. The size of memory and register files is finite and is based on the number of times the memory/register file is accessed [5, 8, 10]. The finite domain property is then used to find the bit-width of scalar variables. For synthesizing a VHDL model of the design, the user needs to do the following steps,

1. Create a netlist from the formula
2. Set the size of register files and memory

3. Determine the bit width of scalars
4. Print the VHDL

A sample Tcl script to generate the VHDL for the trivial processor depicted in Figure 1 is given below.

```
# Synthesizing the design into VHDL

# we need to first create a Netlist from the formula

set spec_net [Net_CreateFromFormula [$spec_regfile cget -raw]]
# Set size of regfile (experimenting with a size of 8)

Net_SetMemSize $spec_net "mem@specRF" 8

# Set width of the scalars
# Since this is an experiment, setting an arbitary width

set scalarWidth 4

# set VHDL filename

set vhdlFile "spec.vhdl"

# set the entity name

set entityName ‘‘spec"

# print the VHDL

Net_PrintVHDL $spec_net $scalarWidth $vhdlFile $entityName
```

3.3 Design Verification

Tarski provides several methods to analyze and simplify designs. The best way to learn how to verify designs using Tarski is to learn through examples. Let us consider the processor described in Section 2. We need to verify that that the two models (single cycle and 2-stage pipeline) are equivalent to each other. A sample Tcl script to verify the design using the approach in [6] is given below.

```
# Declaration of constants

set false [FALSE]
set true [TRUE]

# Define the specification and implementation

source dlx-bd-spec.tcl
source dlx-bd-impl.tcl

# We are following the Burch and Dill approach
# to verifying the trivial processor
```

```
# Step1 -- Project implementation onto the the specification

# First, we need to flush the implementation pipeline
# this is done by making the impl_stall input true

set flush [TRUE]

# Constrain the impl_stall input in implementation to be TRUE

Net_ConstrainInput $impl_net "impl_stall" [$flush cget -raw]

# Now unfold the implementation by two steps
# to enable projection of implementation

set impl_unfolded [node_tfe [$impl_regfile cget -raw] 2]

# Unconstrain the impl_stall input in implementation

Net_UnConstrainInput $impl_net "impl_stall" [$flush cget -raw]

# We must now execute one instruction in both implementaion and spec
# This can be done by constraining stall inputs in both models
# with a register whose initial state is FALSE and next state is TRUE

set flush_reg [br "flush_reg"]
NodeAssignRegister $flush_reg $false $true

Net_ConstrainInput $impl_net "impl_stall" [$flush_reg cget -raw]
Net_ConstrainInput $spec_net "spec_stall" [$flush_reg cget -raw]

# Check whether spec_regfile and impl_regfile are equivalent

set spec_neq_impl [[$spec_regfile equal $impl_regfile] inv]

# Execute a single instruction through both spec and impl
# and compare the results. This can be done by unfolding
# spec_neq_impl for 3 steps (hint: 2 stage pipeline)

set neq_unfolded [node_tfe [$spec_neq_impl cget -raw] 3]

# Simplify the unfolded instance
set neq_unfolded [Net_SimplifyUsingConstantPropagation \
                  $neq_unfolded]

# Project implementation onto specification by constraining cycle 0
# instance of specinitRF. We need to first build a Netlist

set neq_unfolded_net [Net_CreateFromFormula $neq_unfolded]
Net_ConstrainInput $neq_unfolded_net "specinitRF_0" $impl_unfolded

# Now we need to do Read after Write transformation
```

```
set new_neq_unfolded_net [Net_CreateFromFormula $neq_unfolded]
Net_FreeNet $neq_unfolded_net
set neq_unfolded_net [Net_ReplaceWritewithRead \
               $new_neq_unfolded_net "mem\#write" "mem@read" 0]

# Do more simplification and remove UIFs

set new_neq_unfolded [Net_AccessOutput $neq_unfolded_net 0]
set new_neq_unfolded [Net_SimplifyUsingConstantPropagation \
                      $new_neq_unfolded]
set netWithUIFs [Net_CreateFromFormula $new_neq_unfolded]
Net_DoBryantReduction $netWithUIFs
set neq_output [Net_AccessOutput $netWithUIFs 0]
set netWithoutUIFs [Net_CreateFromFormula $neq_output]

# View the final Netlist using dot

Net_PrintFormulaWrap $netWithoutUIFs $neq_output $printFile
source create_dot.tcl

# Synthesize the VHDL

# Analyze the Netlist to determine the bit width of Scalars

set scalarWidth [Net_AnalyzeScalarBitWidth $netWithoutUIFs]

# set up the entity name and name of VHDL file

set entity "dlxbd"
set vhdlFile "dlxbd.vhdl"

# print the VHDL

Net_PrintVHDL $netWithoutUIFs $scalarWidth $vhdlFile $entity
```

3.3.1 Sequential Modeling

The second approach to verify the equivalence of the two models is depicted below (sequential modeling is used here).

```
set false [FALSE]
set true [TRUE]

# Define the specification and implementation

source dlx-bd-spec.tcl
source dlx-bd-impl.tcl

# We are going to verify the processors in the sequential domain
# The idea is to execute an arbitary sequence of two instructions
```

```
# thru both spec and impl and verify that the spec and impl
# register files are equiv after instr sequence has been executed

# We will constrain the stall inputs appropriately to ensure
# only two instr are executed in both spec and impl

set flush_reg1 [br "flush_reg1"]
set flush_reg2 [br "flush_reg2"]

NodeAssignRegister $flush_reg1 $false $flush_reg2
NodeAssignRegister $flush_reg2 $false $true

Net_ConstrainInput $spec_net "spec_stall" [$flush_reg1 cget -raw]
Net_ConstrainInput $impl_net "impl_stall" [$flush_reg1 cget -raw]

# Constrain init_bubble_ex to false
Net_ConstrainInput $impl_net "init_bubble_ex" [$true cget -raw]

# Initialize initial value of spec to be the same as impl
Net_ConstrainInput $spec_net "specinitRF" \
                   [$init_impl_regfile cget -raw]

# Need to do analysis to figure out the bit width of scalars

set spec_neq_impl [[$spec_regfile equal $impl_regfile] inv]

# 4 step unfolding to enable execution of 2 instructions

set neq_unfolded [node_tfe [$spec_neq_impl cget -raw] 4]

# Simplify the unfolded instance and create the netlist

set neq_unfolded [Net_SimplifyUsingConstantPropagation \
                  $neq_unfolded]
set neq_unfolded_net [Net_CreateFromFormula $neq_unfolded]

set scalarWidth [Net_AnalyzeScalarBitWidth $neq_unfolded_net]

# Figure out when impl has committed the sequence of 2 instrs

set impl_commit [$bubble_ex & $flush_reg1]

# set up a random read address

set rand_address [si "rand_read_address"]

# get the data from regfiles for the random address

set spec_rand_regfile_data [uif2 "mem@read" $spec_regfile \
                            $rand_address]
set impl_rand_regfile_data [uif2 "mem@read" $impl_regfile \
                            $rand_address]
```

```
set data_diff [ $impl_commit & [ [$spec_rand_regfile_data equal \
                $impl_rand_regfile_data] inv ]]
set diff_net [Net_CreateFromFormula [$data_diff cget -raw]]

set vhdlFile "dlxseq.vhdl"

set entity "dlxseq"

# print the VHDL

Net_PrintVHDL $diff_net $scalarWidth $vhdlFile $entity
```

4 Installing Tarski

Tarski is available for free download from `http://tarski.sourceforge.net`. Users need to install the following software to build and run Tarski.

1. Tarski uses SWIG to build the Tcl interface. You need to install SWIG to build Tarski (`http://www.swig.org/download.html`)
2. You need to have Tcl 8.4 to build and run Tarski (`http://tcl.sourceforge.net`)
3. To visualize the Netlists, you need to download Graphviz (`http://www.graphviz.org/download`)

4.1 Install Directory Structure

The installation directory has the following structure:

- **util** – Contains libraries (libcudd.a, libcu.a, libutil.a) that need to linked to build Tarski
- **include** – Contains header files to enable compilation
- **vhdl_database** – Database of parameterized VHDL files for modeling memories and uninterpreted functions.
- **tcl_testcases** – Set of sample Tcl scripts
- **micro_arch.c dlx_deep_pipeline.c fold_unfold.c dlx-uclid.c** – Benchmarks specified in C++
- **create_dot.tcl** – Tcl script to generate pdf from dot output

References

[1] W. Ackermann. Solvable cases of the decision problem. In *Studies in Logic and the Foundations of Mathematics*, 1954.

[2] Jason Baumgartner. *Automatic Structural Abstraction Techniques for Enhanced Verification*. PhD thesis, University of Texas, Dec. 2002.

[3] R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Computer-Aided Verification*, 1999.

[4] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. In *ACM Transactions on Computational Logic*, 2001.

[5] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic and lambda expressions and uninterpreted functions. In *Computer-Aided Verification*, 2002.

[6] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In *Computer-Aided Verification*, 1994.

[7] N. Een, N. Sorensson, and A. Mishchenko. Applying logic synthesis for speeding up SAT. In *International Conference on Theory and Applications of Satisfiability Testing*, 2007.

[8] Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Verification of embedded memory systems using efficient memory modeling. In *DATE*, 2005.

[9] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In *Computer-Aided Verification*, 1998.

[10] Panagiotis Manolios, Sudarshan K. Srinivasan, and Daron Vroon. Automatic memory reductions for rtl model verification. In *International Conference on Computer-Aided Design*, 2006.

[11] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[12] Hari Mony, Jason Baumgartner, Viresh Paruthi, Robert Kanzelman, and Andreas Kuehlmann. Scalable automated verification via expert-system guided transformations. In *FMCAD*, Nov. 2004.

[13] Amir Pnueli, Ofer Shtrichman, and Michael Siegel. Translation validation for synchronous languages. In *International Colloqium on Automata, Languages, and Programming*, 1998.

[14] R. E. Shostak. A practical decision procedure for arithmetic with function symbols. In *Journal of the ACM*, 1979.