Proof Engineering

How to formally verify a piece of hardware in a reasonable amount of time with interactive theorem proving

Kenneth Roe
September 24th, 2020
How verification is done today

Simulation—slow and only covers limited cases

Model checking provides broader case coverage but still has exponential behavior with respect to the size of a design
Interactive theorem proving is tedious but resolves model checking complexity issues.

Study methods to improve proof development productivity. We should measure a proof productivity ratio—a ratio between the time needed to verify the design over the time needed to develop the design.
How ITP eliminates complexity
How ITP eliminates complexity

- Interactive theorem proving allows development of the concept of a domino covering one black and one white square
- Proof can easily be generalized to any size chess board
  - 10x10
  - 100x100
  - 10000x10000
- Model checking does search of all possible layouts
  - Learning used to prune search space
How ITP eliminates complexity
Interactive Theorem Proving

- Benefits
  - Potential to eliminate complexity
  - Potential to verify 100% of inputs and components
- Requires development of proofs
  - Model Checking is automatic
- This talk discusses engineering large proofs
Organization

1) Integrating with traditional workflows
2) Best Practices
   a) Structured unfolding
   b) Packaging lemmas with definitions
3) Rewriting
   a) autorewrite vs simpl or compute
   b) Reflection
   c) ml-plugin rewriting package
4) Workflows and CoqPIE
Integrating ITP into existing workflows

1. Develop System Verilog Model and System Verilog Assertions
2. Perform initial wiggling with traditional model checker (such as Jasper)
3. When a property gets too complex, generate proof setup for ITP
4. Repeat until success:
   a. Attempt to prove property
      i. Unlike a model checker, this process is not automatic
   b. If the proof is successful, then EXIT LOOP AND WE ARE DONE
   c. Most likely, a revision to the System Verilog Model will be necessary
   d. Regenerate proof setup
   e. Replay and revise work from step 4a
Interactive theorem proving framework

- System Verilog Model
- SV to Theorem Prover
- Theorem to prove and Translated model
- User developed proof
- Theorem Prover IDE

System Verilog Semantics

Theorem prover (Coq/Isabelle/Lean/ACL2/PVS/HOL)
The System Verilog model

- Provides a mathematically precise definition of System Verilog
- Will resolve many ambiguities in published spec
- Recommended to be donated to Accellera
Organization

1) Integrating with traditional workflows
2) Best Practices
   a) Structured unfolding
   b) Packaging lemmas with definitions
3) Rewriting
   a) autorewrite vs simpl or compute
   b) Reflection
   c) ml-plugin rewriting package
4) Workflows and CoqPIE
Kami language

- HDL built on top of Coq
- Modules
  - Registers
    - Name and type (bit width)
  - Rules
    - Name and code
    - Code is both condition and action of rule
  - Methods
    - Called by rules
    - Name and code
- Rule/method code
  - Imperative language
    - If, Let, Register read, Register write, Method call, Module calls, Debug print
  - Compiles to boolean logic
- Gallina functions serve as meta language to generate Kami
Proving WfMod--A case study

WfMod ty (processor func_units deviceTree memParams)
What doesn't work--unfolding everything

unfold processor.
unfold processorCore.
unfold debugInterruptRule.
unfold completionBufferFetcherCompleteRule.
unfold ProcessorCore.pipeline.
unfold trapInterruptRule.
unfold impl.

.
What doesn't work--unfolding everything

WfMod ty
  (ConcatMod
   (makeModule
    (MERegister
     (^ ("mode"),
      existT RegInitValT (SyntaxKind PrivMode)
      (Some (makeConst $ (MachineMode)%word))))] ++
...
(about 1000 lines more)
What doesn't work--unfolding everything

Local Definition printFuncUnitInstName (fu: FuncUnitId @# ty) (inst: InstId @# ty): ActionT ty Void :=
(GatherActions (map (fun i =>
  If ($ (fst i) == fu)
    then (System [DispString _ (fuName (snd i)); DispString _ "."];)"
    (GatherActions (map (fun j =>
      If ($ (fst j) == inst)
        then (System [DispString _ (instName (snd j))]; Retv)
        else Retv; Retv) (tag (fulnsts (snd i))) as _; Retv))
    else Retv; Retv) (tag func_units)) as _; Retv)%kami_action.
Unfolding best practices

1) Only unfold one or two levels of definitions
   a) Try and keep goals small
2) Introduce lemmas to reason about deeper levels
Unfolding best practices--benefits

1) Reduces complexity
   a) Multiple instances of a function are only reasoned about once
   b) Coq theorem is much faster
2) Makes proofs more modular
   a) Easier to update if the design changes
3) Makes proofs more readable
Structured unfolding example

Goal:
\[ \text{WfMod ty (processor func_units deviceTree memParams)} \]

\text{unfold processor.}
Structured unfolding example

\[
\text{WfMod ty} \\
\text{(createHideMod} \\
\text{(ConcatMod (processorCore func_units deviceTree memParams)} \\
\text{(deviceMod deviceTree} \\
\text{(Ifc.ArbiterTag} \\
\text{(ProcessorCore.pipeline func_units deviceTree memParams))))} \\
\text{)(map fst} \\
\text{(getAllMethods} \\
\text{(ConcatMod (processorCore func_units deviceTree memParams)} \\
\text{(deviceMod deviceTree} \\
\text{(Ifc.ArbiterTag} \\
\text{(ProcessorCore.pipeline func_units deviceTree memParams))))}))\].

apply \text{WfModCreateHideMod}.\]
Structured unfolding example

```
SubList
  (map fst ...)
  (map fst ...) \/
WfMod ty
  (ConcatMod (processorCore func_units deviceTree memParams)
    (deviceMod deviceTree
     (Ifc.ArbiterTag
      (ProcessorCore.pipeline func_units deviceTree memParams))))

split.  apply SubList_refl.  apply concatModWf;...
```
Structured unfolding example

7 subgoals that look like this:

DisjKey (getAllRegisters (processorCore func_units deviceTree memParams))
(getAllRegisters
 (deviceMod deviceTree
  (Ifc.Arbitrator
   (ProcessorCore.pipeline func_units deviceTree memParams)))))
Structured unfolding example

Introduce auxiliary theorem.

Theorem DisjKey_getAllRegisters_processorCore_deviceTree:
DisjKey (getAllRegisters (processorCore func_units deviceTree memParams))
getAllRegisters
  (deviceMod deviceTree
   (Ifc.ArbiterTag (ProcessorCore.pipeline func_units deviceTree memParams))).
Organization

1) Integrating with traditional workflows
2) Best Practices
   a) Structured unfolding
   b) Packaging lemmas with definitions
3) Rewriting
   a) autorewrite vs simpl or compute
   b) Reflection
   c) ml-plugin rewriting package
4) Workflows and CoqPIE
Local Definition tokenStartRule: ActionT ty Void :=
Read initReg <- @"initReg";
If !#initReg
then (System [DispString _ "[tokenStart]\n"];
LET tokenVal: Void <- $(getDefaultConst Void);
LETA _ <- @Fifo.Ifc.eng _ tokenFifo _ tokenVal;
Write @"initReg" <- $true;
Retv );
Retv.
Theorem \texttt{WfActionT\_new}\_tokenStartRule:

\begin{verbatim}
  \forall \texttt{ty}, @WfActionT\_new\_ty
  ((\@^ ("initReg"), \texttt{existT RegInitValT (SyntaxKind Bool) (Some (SyntaxConst false)))::(Fifo.Ifc.regs Impl.tokenFifo))
  Void (Pipeline.Ifc.tokenStartRule (impl func_units
deviceTree memParams))).
\end{verbatim}

Proof.

\begin{verbatim}
 ...
\end{verbatim}

Hint Rewrite \texttt{WfActionT\_new\_tokenStartRule : WfActionT\_new\_reduce}. 
Organization

1) Integrating with traditional workflows
2) Best Practices
   a) Structured unfolding
   b) Packaging lemmas with definitions
3) Rewriting
   a) `autorewrite vs simpl or compute`
   b) Reflection
   c) ml-plugin rewriting package
4) Workflows and CoqPIE
autorewrite vs. simpl or compute

- simpl and compute only apply definitions when applicable
- compute can introduce extraneous “fix” constructs
- autorewrite applies properties
  - Distribution
  - Lemmas to deal with parameterized subterms
Compute introduces fix constructs

\text{DisjKey (getAllMethods (processorCore func_units deviceTree memParams))}
(\text{getAllMethods}
(\text{deviceMod deviceTree}
(\text{Ifc.ArbiterTag}
(\text{ProcessorCore.pipeline func_units deviceTree memParams}))))

...
Compute introduces fix constructs

DisjKey (getMethods (processorCore func_units deviceTree memParams))

(fix getAllMethods (m : Mod) : list (string * {x : Signature & MethodT x}) :=
match m with
| Base m' => getMethods m'
| HideMeth m' _ => getAllMethods m'
| ConcatMod m1 m2 => getAllMethods m1 ++ getAllMethods m2
end)

(deviceMod deviceTree
 (Ifc.ArbiterTag
   (ProcessorCore.pipeline func_units deviceTree memParams))))
autorewrite vs. simpl or compute

DisjKey

[[[@^ "_debugMode")%string, existT (fun x : FullKind => RegInitValT x) (SyntaxKind Bool) ?Goal2]]

((@^ ("mode"), existT RegInitValT (SyntaxKind PrivMode) (Some (makeConst $(MachineMode)%word)));
 ::(@^ ("reservation"), existT RegInitValT (SyntaxKind (Maybe Reservation)) (Some (makeConst Default)));
 :: csr_regs Csrs ++
 ( [[@^ ("initReg"), existT RegInitValT (SyntaxKind Bool) (Some (SyntaxConst false))];  
   (@^ ("pc"), existT RegInitValT (SyntaxKind (Bit Xlen)) (Some (SyntaxConst pcInit))))
)]
autorewrite vs. simpl or compute

Rewrite rules for evaluation

\[ \text{DisjKey list1 list2} = \forall x, \text{In x (map fst list1)} \rightarrow \text{In x (map fst list2)} \rightarrow \text{False} \]

\[ \forall f \ a \ b, \text{map f (a::b)} = (f \ a)::(\text{map f b}) \]
\[ \forall f, \text{map f []} = [] \]

\[ \forall a \ b, \text{fst (a,b)} = a \]

\[ \text{In x nil} = \text{False} \]
\[ \text{In x (a::b)} = ((x=a) \lor \text{In x b}) \]

...
auto rewrite vs. simpl or compute

```ml
forall x,
In x
.map fst
[((@^ "_debugMode")%string, existT (fun x : FullKind => RegInitValT x) (SyntaxKind Bool) ?Goal2)]
-
> In x
(map fst 
((@^ ("mode"), existT RegInitValT (SyntaxKind PrivMode) (Some (makeConst $(MachineMode)%word))
::(@^ ("reservation"), existT RegInitValT (SyntaxKind (Maybe Reservation)) (Some (makeConst Default))))
:: csr_regs Csrs ++
((@^ ("initReg"), existT RegInitValT (SyntaxKind Bool) (Some (SyntaxConst false));
(@^ ("pc"), existT RegInitValT (SyntaxKind (Bit Xlen)) (Some (SyntaxConst pcInit))))))
> False```
autorewrite vs. simpl or compute

forall x,
  x=(@^"_debugMode")%string
->
x=(@^("mode")) V x=(@^("reservation")) R
  ln x (map fst (csr_regs CSRS++
    ([(^"initReg"), existT RegInitValT (SyntaxKind Bool) (Some (SyntaxConst false)));
    (^["pc"), existT RegInitValT (SyntaxKind (Bit Xlen)) (Some (SyntaxConst pcInit)))]))
-> False
autorewrite vs. simpl or compute

Useful distribution properties

forall f a b, map f (a++b)=(map f a)++(map f b)
forall x a b, In x (a++b)=(In x a \union In x b)
...

autorewrite vs. simpl or compute

forall x,
  x=(^ "_debugMode")%string
->
  x= (^("mode")) \ x=(^("reservation"))\/
    \n    In x (map fst ( csr_regs CSRS)) \+
    (map fst ([(^("initReg"), existT RegInitValT (SyntaxKind Bool) (Some (SyntaxConst false)));
               (^("pc"), existT RegInitValT (SyntaxKind (Bit Xlen)) (Some (SyntaxConst pcInit)))]))
-> False
autorewrite vs. simpl or compute

forall x,
  x=(@^ "_debugMode")%string
->
  x= (@^("mode")) ∨ x=(@^("reservation"))\/
  In x ((map fst (csr_regs CSRS))++
       [[@^("initReg"),[@^("pc")]]])
-> False
autorewrite vs. simpl or compute

forall x,

\[ x = (@^ "\_debugMode") % \text{string} \]

\[ \rightarrow \]

\[ x = (@^("mode")) \lor x = (@^("reservation")) \lor \]

\[ \text{let } x = (@^("initReg")) \land x = (@^("pc")) \]

\[ \rightarrow \text{False} \]
autorewrite vs. simpl or compute

H: x=(@^ "_debugMode")%string
H0: In x ((map fst (csr_regs CSRS)) =============
False
Speeding up Rewriting

- autorewrite too slow
  - Issue is unique to Coq
  - Other theorem provers such as ACL2, Isabelle and Lean have better performance
- Two solutions
  - Reflection
    - Proofs verified with respect to Coq Kernel
    - 10x speedup
  - ml-plugin
    - Proofs not verified with respect to Coq Kernel
    - Need to trust ml-plugin
    - 100x speedup
    - Many specialized algorithms
Organization

1) Integrating with traditional workflows
2) Best Practices
   a) Structured unfolding
   b) Packaging lemmas with definitions
3) Rewriting
   a) autorewrite vs simpl or compute
   b) Reflection
   c) ml-plugin rewriting package
4) Workflows and CoqPIE
Reflection

- Create a Coq data type representing Coq expressions
- Reify Coq expression to the data type
- Write a recursive function to simplify the data type representation
- Denote data type representation back to a Coq expression
Organization

1) Integrating with traditional workflows
2) Best Practices
   a) Structured unfolding
   b) Packaging lemmas with definitions
3) Rewriting
   a) autorewrite vs simpl or compute
   b) Reflection
   c) ml-plugin rewriting package
4) Workflows and CoqPIE
ML plugin overview

1. Coq expression converted to rewriter expression
2. Expression simplified
3. Rewriter expression converted back to Coq expression
ML plugin is a composition of algorithms

- Top level algorithm is an inner rewriting algorithm
  - All subterms simplified first
  - Then top level term simplified
  - The whole process is repeated if anything changed
- Many different simplification algorithms applied
Features

- Recursive path ordering to ensure termination
- Contextual rewriting
- Arithmetic builtins
- Quantifier simplifications
- Higher order functions
- AC handling
- Total orders/partial orders
  - Transitive chains
Rewriting comparisons

- ml-plugin rewriting library
  \[ f(f(x)) = x \land f(f(f(x))) = x \]
- Isabelle
  \[ f(f(x)) = x \land f(f(f(x))) = x \]
- Coq
  \[ f(f(x)) = x \land f(f(f(x))) = x \]
Rewriting comparisons

- ml-plugin rewriting library
  \[ f(f(x))=x \land f(f(f(x)))=x \implies f(x)=x \] 😊

- Isabelle
  \[ f(f(x))=x \land f(f(f(x)))=x \implies \text{fails} 😞

- Coq
  \[ f(f(x))=x \land f(f(f(x)))=x \implies f(f(x))=x \land f(f(f(x)))=x 😞 \]
Rewriting comparisons

- ml-plugin rewriting library
  $\exists x. f(x) = g(4) \land x = 3$
- Isabelle
  $\exists x. f(x) = g(4) \land x = 3$
- Coq
  $\exists x. f(x) = g(4) \land x = 3$
Rewriting comparisons

- ml-plugin rewriting library
  \[ \exists x. f(x) = g(4) \land x = 3 \implies f(3) = g(4) \]

- Isabelle
  \[ \exists x. f(x) = g(4) \land x = 3 \implies f(3) = g(4) \]

- Coq
  \[ \exists x. f(x) = g(4) \land x = 3 \implies \exists x. f(x) = g(4) \land x = 3 \]
Rewriting comparisons

- ml-plugin rewriting library
  \(5 < y \land y < z \land z < 3\)
- Isabelle
  \(5 < y \land y < z \land z < 3\)
- Coq
  \(5 < y \land y < z \land z < 3\)
Rewriting comparisons

- ml-plugin rewriting library
  \[ 5 < y \land y < z \land z < 3 \Rightarrow \text{False} 😞 \]
- Isabelle
  \[ 5 < y \land y < z \land z < 3 \Rightarrow \text{False} 😞 \]
- Coq (omega fails as well, only works if the result is True)
  \[ 5 < y \land y < z \land z < 3 \Rightarrow 5 < y \land y < z \land z < 3 \]
Rewriting comparisons

- ml-plugin rewriting library
  \[ \forall x, 2 \times x + 1 = 7 \]
- Isabelle
  \[ \forall x, 2 \times x + 1 = 7 \]
- Coq
  \[ \forall x, 2 \times x + 1 = 7 \]
Rewriting comparisons

- ml-plugin rewriting library
  \[ \forall x, 2 \times x + 1 = 7 \Rightarrow \forall x, x = 3 \smiley \]

- Isabelle
  \[ \forall x, 2 \times x + 1 = 7 \Rightarrow \text{fails} \frown \]

- Coq
  \[ \forall x, 2 \times x + 1 = 7 \Rightarrow \forall x, x + (x+0) + 1 = 7 \]
Rewriting comparisons

- ml-plugin rewriting library
  \[ x+y=0 \rightarrow y+3+x=q \]
- Isabelle
  \[ x+y=0 \rightarrow y+3+x=q \]
- Coq
  \[ x+y=0 \rightarrow y+3+x=q \]
Rewriting comparisons

- ml-plugin rewriting library
  \[ x+y=0 \rightarrow y+3+x=q \] => \[ 0=y+x\rightarrow 3=q \] 😊

- Isabelle
  \[ x+y=0 \rightarrow y+3+x=q \] => fails 😞

- Coq
  \[ x+y=0 \rightarrow y+3+x=q \] => \[ x+y=0 \rightarrow y+3+x=q \] 😞
Organization

1) Integrating with traditional workflows
2) Best Practices
   a) Structured unfolding
   b) Packaging lemmas with definitions
3) Rewriting
   a) autorewrite vs simpl or compute
   b) Reflection
   c) ml-plugin rewriting package
4) Workflows and CoqPIE
Theorem proving workflows

- Replay important
  - Often proof statements change
  - Steps need to be adjusted
- Many opportunities to automatically generate proofs
  - `WfActionT_new_…` theorems an example
- Need to be able to quickly see changes in a goal
- Browsing of proofs
CoqPIE

- CoqPIE parser
  - Simplified version of syntax
  - Both source files and Goal state output parsed
- Project pre-compiled and intermediate goals saved
- Advanced features
  - Replay
  - Lemma extraction
- Future features
  - Lemma generation/editing
    - Eg. WfActionT_new… series
Conclusions

1) Proof engineering is key to making interactive theorem proving work
   a) Best practices
   b) Rewriting
   c) Workflows and CoqPIE
2) Interactive theorem proving has the potential of providing a much more powerful verification methodology than model checking
Websites

http://www.cs.jhu.edu/~roe
   The author’s PhD thesis and other links
https://github.com/sifive/RiscvSpecFormal
   Kami/ProcKami
https://github.com/kendroe/CoqRewriter
   Coq rewriter ml-plugin
https://github.com/kendroe/coqpie
   CoqPIE IDE