A Toolbox for Property Checking from Simulation using Incremental SAT

Rob Sumners
Centaur Technology
rsumners@centtech.com

...terrible title...

Rob Sumners
Centaur Technology
rsumners@centtech.com

Motivation: “extending” simulation..

- Hardware designs go through extensive verification via design simulation.
- Verification engineers define simulation environments which:
  - Generate random and/or directed legal input stimulus.
  - Check that the behavior of the design matches specification.
- These environments are extensive and (unfortunately) often written in “higher-level” languages..
  - These definitions are not finite-state and in general, not amenable to formal tools

Motivation: “extending” simulation..

- Goal: use existing simulations of hardware designs to efficiently search for property failures..
- Simple approach:
  - Take dump file from existing simulation to specify initial state and input sequence..
  - From each state in the simulation, generate a quick search for property failures..
    - A limited, directed form of bounded model checking
- Different approach:
  - Use input transformation to reduce the number of inputs and afford deeper search..
  - Use Incremental SAT and dynamically manage the size of the SAT clauses

Two parts to present..

- First part: a tool that extends simulation with limited efficient search for failure using..
- Second part: a core (or "toolkit") for efficiently checking properties of a finite next-state function evaluated over time
    - Uses incremental SAT (via books/centaur/ipasir interface) to amortize the cost of SAT queries across sequential clocks

First part: tool steps..

- Read in design (in Verilog)
  - Process from top module to build an AIGNET for next-state
  - Makes use of VL, SV, etc. to translate to SVEX, then AIGs, then AIGNET.
- Read in VCD file dumped from simulation
  - Process to get initial state and input values at each clock cycle
- Initialize and Execute main search loop
  - ...more on this in a moment..
- Report results back to the user
  - ..including generation of VCD in case of failure

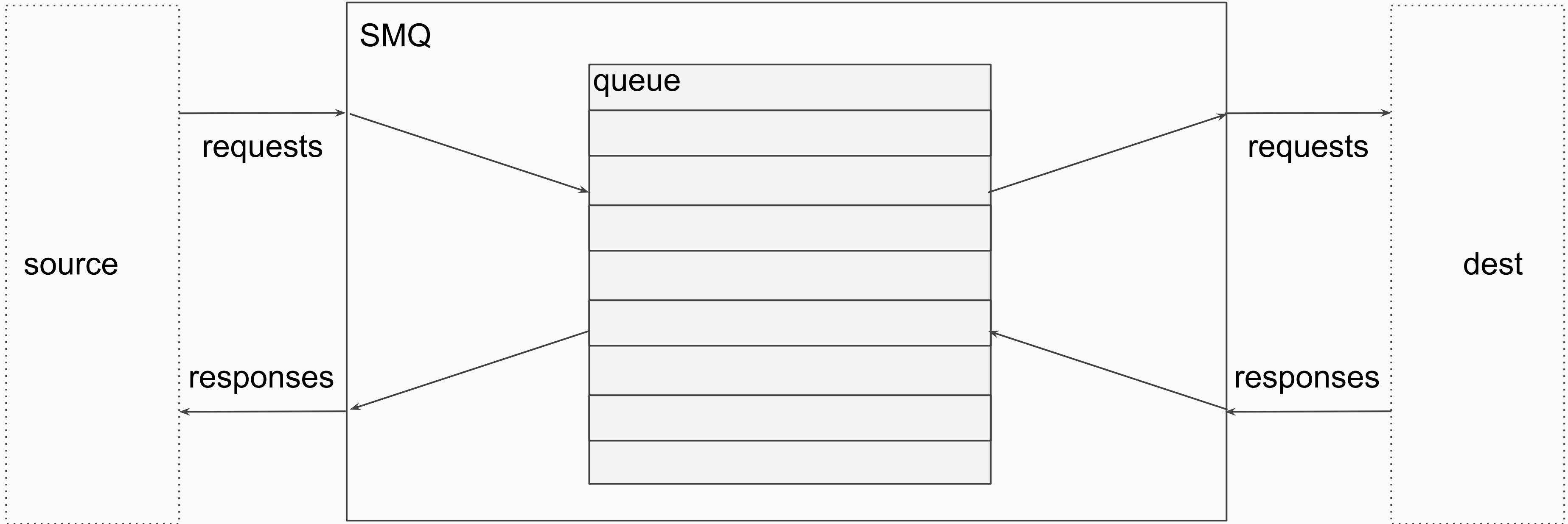
First part: main search loop..

- Iterates a “search window” which defines the current property check.
- Repeat until pass/fail:
  - **Choose step** .. determine which step to take:
    - Heuristically choose which step to take next based on current targets, search measurements, and recent step results
  - **Take step** .. pick one of the following:
    - Grow-window -- add clauses for checking the next property in time
    - Shrink-window -- add unit clauses for reducing based on input binding
    - Check-fails -- SAT check the fails in the current search window
    - Cleanup-state -- cleanup memory and (optionally) reset incremental SAT

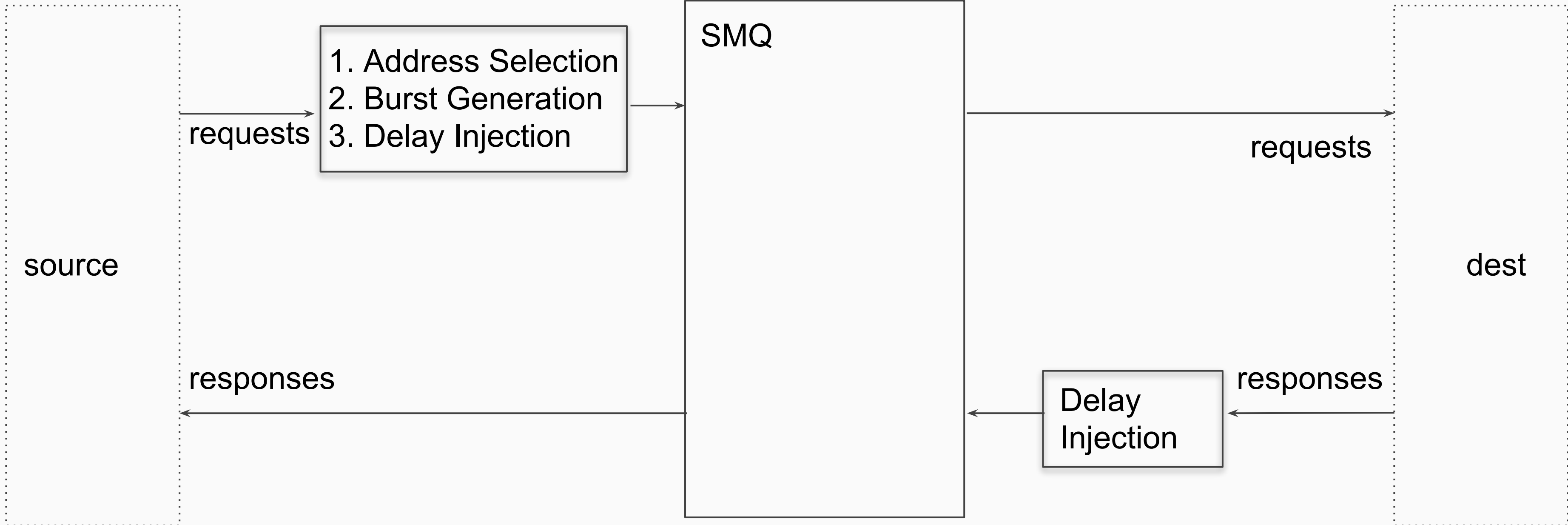
Example: SMQ simple memory queue

- Simple read-write queue with:
  - Read and write requests coming from **source** port and forwarded to **dest** port
  - Responses flow from **dest** port back to **source** port
- Excluding address matches, reads can pass writes except:
  - If enough writes are blocked, then we enter write-burst mode.. writes get priority
  - In addition, the oldest entry can also only be bypassed a maximum number of times
- Property: check that if some entry is always ready to go

Example: SMQ simple memory queue



First part: SMQ adding input transformers..



Example: SMQ Adding Input Transformers

- Transform #1 : address selection
  - Pick focal address, then (based on free input) pick that address
- Transform #2 : burst generation
  - Based on a free input, re-pick the previous command type (read/write in this case)
- Transform #3 : delay injection (on src req. and dest. resp)
  - Free inputs on both src. and resp. to delay delivery (single buffer in this case)
- With these transforms and just allowing fixed random data, the number of free inputs reduced to 4 bits per clock cycle..

Second part: Designing a “core”

- In considering a “proof” of correctness for extended simulation..
- .. a desire arose to design and simplify the “core”
    - Potentially useful for other applications..
- The state of the “core” is comprised of:
    - The AIGNET defining the next-state function of the design
    - The IPASIR stobj storing the state of incremental SAT
    - A 2D-array defining a mapping from design variables (at each clock cycle) to literals
        - Progresses from **bottom** to **SAT literal** to **boolean value**
    - A list of assignments/bindings, a list of constraints, and a list of checked assertions

Second part: Designing the “core”

- The “core” supports the following steps:
  - Initialization
    - take an AIGNET defining next-state function and a fixed number of cycles
    - initialize the state of the “core”
  - Add logic supporting a variable@cycle
    - Variable@cycle goes from **bottom** to **SAT literal**
  - Add constraint or assignments
    - Variable@cycle goes to **boolean value**
  - Check any variable@cycle (must currently map to **SAT literal**)
    - Can add assumptions..

Second part: Proving core invariant

- Define and prove invariant: `(core-inv cs$)`
  - At every step, ensures that satisfying assignments to clauses in IPASIR state are consistent with current constraints, assignments, and definition of next-state via AIGNET given the current binding of variable@cycle to SAT literals.
  - The invariant provides a context for every SAT check performed
    - (assignments and constraints and definitions) => assertion
  - The invariant holds after initialization and is preserved by every other “core” step.

Second part: A few implementation notes..

- Adding assignments causes a fast forward (lifted) evaluation to propagate constants forward..
- Adding the logic supporting a new variable@cycle will only include “cone of influence” based on current variable@cycle map
- In support of this work, we added some additional optional interface functions to the ipasir library to:
  - Get statistics on current SAT clause database state
  - Adjust the priority of certain literals to be chosen
  - These are only defined and will only work for MINISAT-based incremental SAT libraries

Work done and work to be done..

- Have built extended simulation tool as specified
  - Some small-ish examples included in supporting materials
- Have defined and proven a “core” as specified

- Still need to replace “core” in extended simulation tool with proven “core”
- Ongoing work to improve direction and coverage of extended simulation

Questions? .. and answers..

Thank you!!