Incremental SAT Library Integration using Abstract Stobjs

Sol Swords
Centaur Technology, Inc.
ACL2 Workshop 2018

Incremental vs. Monolithic SAT

Monolithic SAT:

- Provide a Boolean formula (CNF), check whether it can be satisfied, exit
- Focus on hard problems and large problems
- Conflict Driven Clause Learning + many preprocessing, inprocessing algorithms

Incremental SAT:

- Check SAT for permanent formula (CNF) and temporary assumption (cube), repeat
- Focus on solving easy problems fast
- Share heuristic info and lemmas between SAT calls
- Mainly uses CDCL -- other procedures possible but less common

Incremental SAT Applications

- SAT Sweeping/FRAIGing: check equivalence of two circuits by repeated SAT checks between candidate equivalences from among internal nodes
  - state of the art Boolean combinational equivalence checking alg.
- IC3/PDR -- state of the art (hardware-oriented) safety model checking
- Bounded model checking
- Many more in hardware verification domain alone
- Max-SAT
- Quantified Boolean Formula solving

Incremental SAT Workflow

1. Create solver object
2. Add clauses to CNF formula --
   (a V b V c) ^ (~a V d V e) ^ …
3. Set a temporary assumption cube -- a ^ ~e ^ g
4. SAT solve for (CNF ^ assumption), assumption is deleted
    a. If SAT, maybe query satisfying assignment
    b. If UNSAT, maybe query unsatisfiable subset of assumption cube
5. Maybe GOTO 2
6. Delete solver.

Why would we want this in ACL2?

Integration of decision procedures with ACL2 has a long & fruitful history:

- ACL2 BDDs
- SULFA
- ACL2SIX
- GL → uBDDs
- GL → AIGs → monolithic SAT via SATLINK
- SMTLINK

- Current main application of incremental SAT in ACL2: SAT sweeping on AIGNET
- Many future possibilities
- Main selling point: Makes it very cheap to call SAT repeatedly on related problems.

Target: IPASIR interface

- Simple C API for incremental SAT
- Used for incremental track in SAT competitions 2015-2017
- 10 functions total…

`ipasir_signature`	Get library version
`ipasir_init, ipasir_release`	Construct/free solver object
`ipasir_add, ipasir_assume`	Set up the formula/temporary assumptions
`ipasir_solve`	Call the SAT solver
`ipasir_val, ipasir_failed`	Post-solve querying
`ipasir_set_terminate`†	Set callback for giving up on solve
`ipasir_set_learn`*	Set callback for learning new clauses

† Partly supported in ACL2

* Not yet supported in ACL2

ACL2 integration approach

- Model the API in ACL2
- Under the hood:
  - Load shared library (using Common Foreign Function Interface)
  - Replace ACL2 API functions with calls to C API through CFFI

Problem: what kind of object is the solver state?

- Need to restrict the API to supported functions
- Non-applicative → must be used single-threadedly

Answer: an abstract stobj!

Abstract Stobj Features

- Single-threaded object with customizable interface and logical model
- Logical model and executable code may be totally different
    - Must preserve some correlation relation to show that execution mirrors logic

For our purposes:

- Single-threadedness enforced for execution
- Can decide on the logical model we want
- Can determine what executable interface functions exist
- Can restrict (using guards) situations in which those interface functions may be used.

Abstract Stobj Contract

For some invariant relation `(corr logic exec)`:

- `(corr (creator-logic) (creator-exec))`
- For each accessor: `(corr logic exec)` → `(equal (acc-logic logic) (acc-exec exec))`
- For each updater: `(corr logic exec)` → `(corr (upd-logic logic) (upd-exec exec))`
- For each interface function: `(corr logic exec) & (guard-logic logic)` → `(guard-exec exec)`

ACL2 requires proof of these properties to admit an abstract stobj.

We can't prove ours (because the `exec` parts aren't defined in the logic). But we argue it anyway...

Soundness Assessment

- We have carefully compared our model with the “contract” of an incremental SAT solver
  - But solvers can still be buggy.
- Other parts of the soundness story
  - Handling nondeterminism -- must not be able to get:
    - two provably equal solver objects
    - a solver object provably equal to one of its previous states
  - Integration artifacts
    - Known soundness bug: can execute redefined interface functions on `ipasir$c` concrete stobj (if you do some work to make them not untouchable).
- More discussion in paper
- Is it sound? Social process of “mathematics” …

Building Formulas in ipasir

- Add literals to build up a clause

  ```
  (ipasir-add-lit lit ipasir),
  (ipasir-finalize-clause ipasir)
  ```

- Add clauses as a whole

  ```
  ipasir-add-unary, ipasir-add-binary, ...,
  ipasir-add-list
  ```

- Build gate constraints -- multiple clauses

  ```
  ipasir-set-and, ipasir-set-xor, ipasir-set-mux
  ```

- Build AIGNET fanin cones -- multiple gates

  ```
  aignet-lit->ipasir, aignet-lit-list->ipasir
  ```

AIGNET to IPASIR

- AIGNET: And/Inverter Graph -- circuit structure -- encoded in stobj array
- `aignet-lit->ipasir` adds CNF to encode the circuit structure in the solver.
- Maintains bidirectional mapping of ipasir literals ↔ aignet literals
- `aignet-lit->ipasir` ensures that the input AIG literal has a corresponding CNF literal

- Maintains invariant: each evaluation of the AIG maps onto a satisfying assignment of the CNF
- Therefore if CNF is UNSAT under some assumptions, the AIG literals corresponding to the assumption cube cannot be simultaneously satisfied → *soundness*

- Conversely, each satisfying assignment of the CNF maps onto an evaluation of the AIG
- Therefore if CNF+assumption is satisfiable, AIG assumption is satisfiable → *completeness*

SAT Sweeping Algorithm

```
fraig(aignet_in)
    map = []; aignet_out = initalize_aignet()
    copy_combinational_inputs(aignet_in, map, aignet_out)
    foreach gate node g = op(a, b) in aignet_in
        copy = find_or_create_gate(op, map[a], map[b], aignet_out)
        candidate = find_possible_equivalent(copy, aignet_out)
        if candidate
            (status, sat_assign) = sat_check_equivalence(copy, candidate, aignet_out)
            case status
                Unsat:  map[g] = candidate
                Sat:    refine_possible_equivalences(sat_assign, aignet_out)
                        map[g] = copy
                Failed: map[g] = copy
        else
            map[g] = copy
    copy_combinational_outputs(aignet_in, map, aignet_out)
```

GL + SAT Sweeping

```
(include-book “centaur/gl/bfr-fraig-satlink” :dir :system)
(include-book “centaur/ipasir/ipasir-backend” :dir :system)
(value-triple (tshell-ensure)) ;; tshell needed for satlink

(gl::gl-simplify-satlink-mode) ;; use AIGs, AIGNET transformations, SAT

(define my-satlink-config ()
   (satlink::make-config ...)) ;; see :doc satlink::config
(defattach gl::gl-satlink-config my-satlink-config)

(define my-transforms-config ()
   ;; see :doc aignet::aignet-comb-transforms
   (list ... (aignet::make-fraig-config ...) ...))
(defattach gl::gl-transforms-config my-transforms-config)

(def-gl-thm ...)
```

Next Steps

- Sequential simplification/model checking algorithms?
- Tighter GL integration?

- Other, non-hardware-specific applications?

- UNSAT proof checking?