Generating Small Countermodels using SMT

Andrew Reynolds
MVD

September 21, 2012

Acknowledgements

- Intel Corporation
  - Amit Goel, Sava Krstic
- University of Iowa
  - Cesare Tinelli, Francois Bobot
- New York University
  - Clark Barrett, Morgan Deters, Dejan Jovanovic

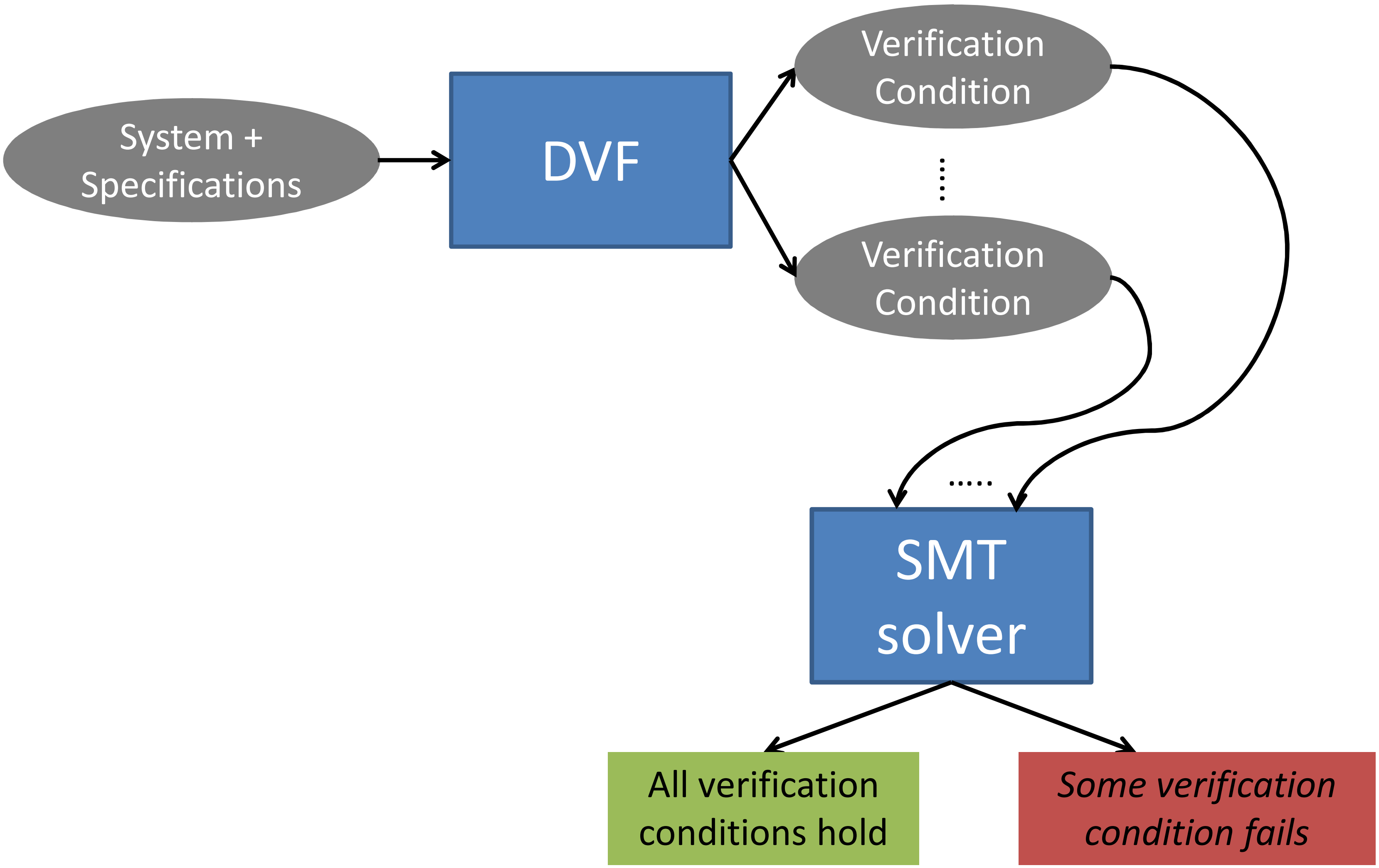
Overview

- Satisfiability Modulo Theories (SMT)
- SMT-Based System Verification
  - Deductive Verification Framework (DVF)
- Challenge of quantifiers in SMT
  - Why do we care about quantifiers?
  - Why are quantifiers difficult?
- Finite Model Finding
- Experimental Results

Satisfiability Modulo Theories (SMT)

- SMT solvers:
  - Are powerful tools for determining satisfiability of ground formulas
    - Built-in decision procedures for many theories
      - Arithmetic, arrays, bit vectors, datatypes, …
  - Have improved performance in past 10 years
- Verification applications rely on SMT solvers
  - System verifier DVF used by Intel

SMT-Based System Verification



DVF Example

Definitions

```
type resource
const resource null
type process
var array(resource, bool) valid = mk_array[resource](false)
var array(resource, int) count
var array(process, resource) ref = mk_array[process](null)
…
module S = Set<type process>
```

Transition System

```
transition create (resource r)
require (r != null, !valid[r]){
  valid[r] := true;
  count[r] := 0;
}
…
```

Properties

```
def bool prop = forall (process p) (ref[p] != null => valid[ref[p]])
def bool refs_non_zero = forall (process p) (ref[p] != null => count[ref[p]] > 0)
…
```

Goals

```
goal main = invariant prop assuming refs_non_zero
…
goal rnz  = formula (… && prop && … => refs_non_zero)
```

- Language corresponds closely to SMT constraints

DVF SMT Backend

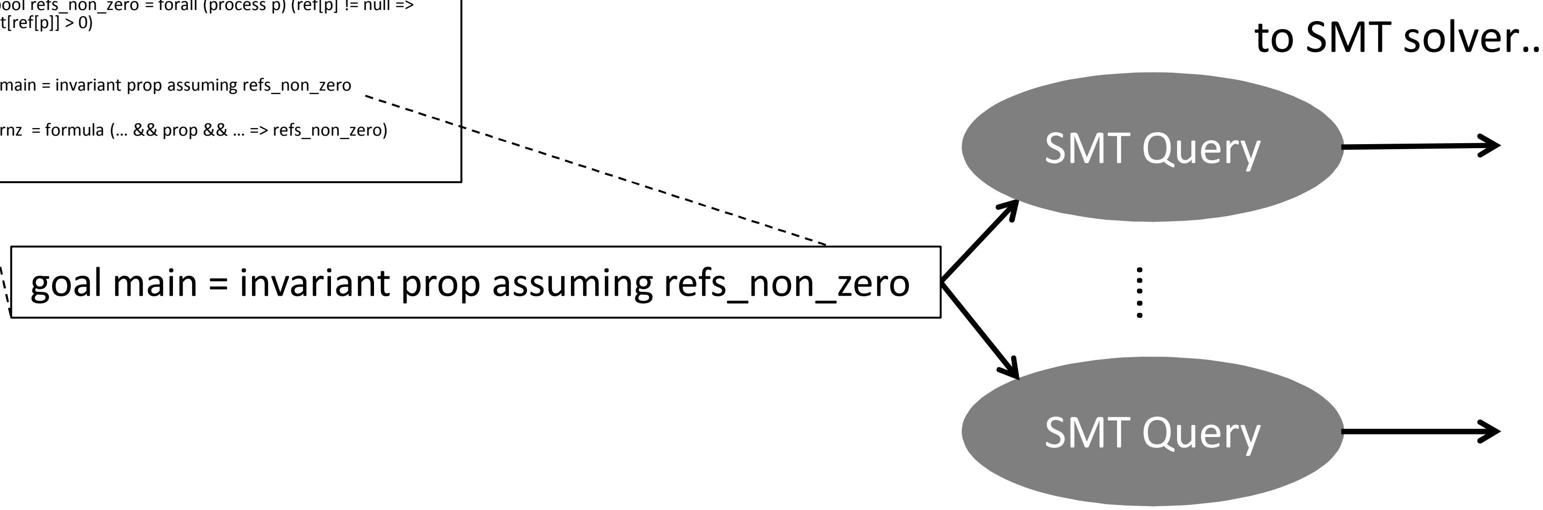
```
type resource
const resource null
type process
var array(resource, bool) valid = mk_array[resource](false)
var array(resource, int) count
var array(process, resource) ref = mk_array[process](null)
…
module S = Set<type process>

transition create (resource r)
require (r != null, !valid[r]){
  valid[r] := true;
  count[r] := 0;
}
…

def bool prop = forall (process p) (ref[p] != null => valid[ref[p]])
def bool refs_non_zero = forall (process p) (ref[p] != null =>
count[ref[p]] > 0)
…

goal main = invariant prop assuming refs_non_zero
…
goal rnz  = formula (… && prop && … => refs_non_zero)
```

Goals



- Goals translated into (possibly multiple) SMT queries

SMT Query

Definitions:

S, P, R : type
null : R
valid: Array(R, Bool)
count: Array(R, Int)
ref: Array(P, R)
empty : S
mem : (S, P) -> Bool
add, remove : (S, P) -> S
…

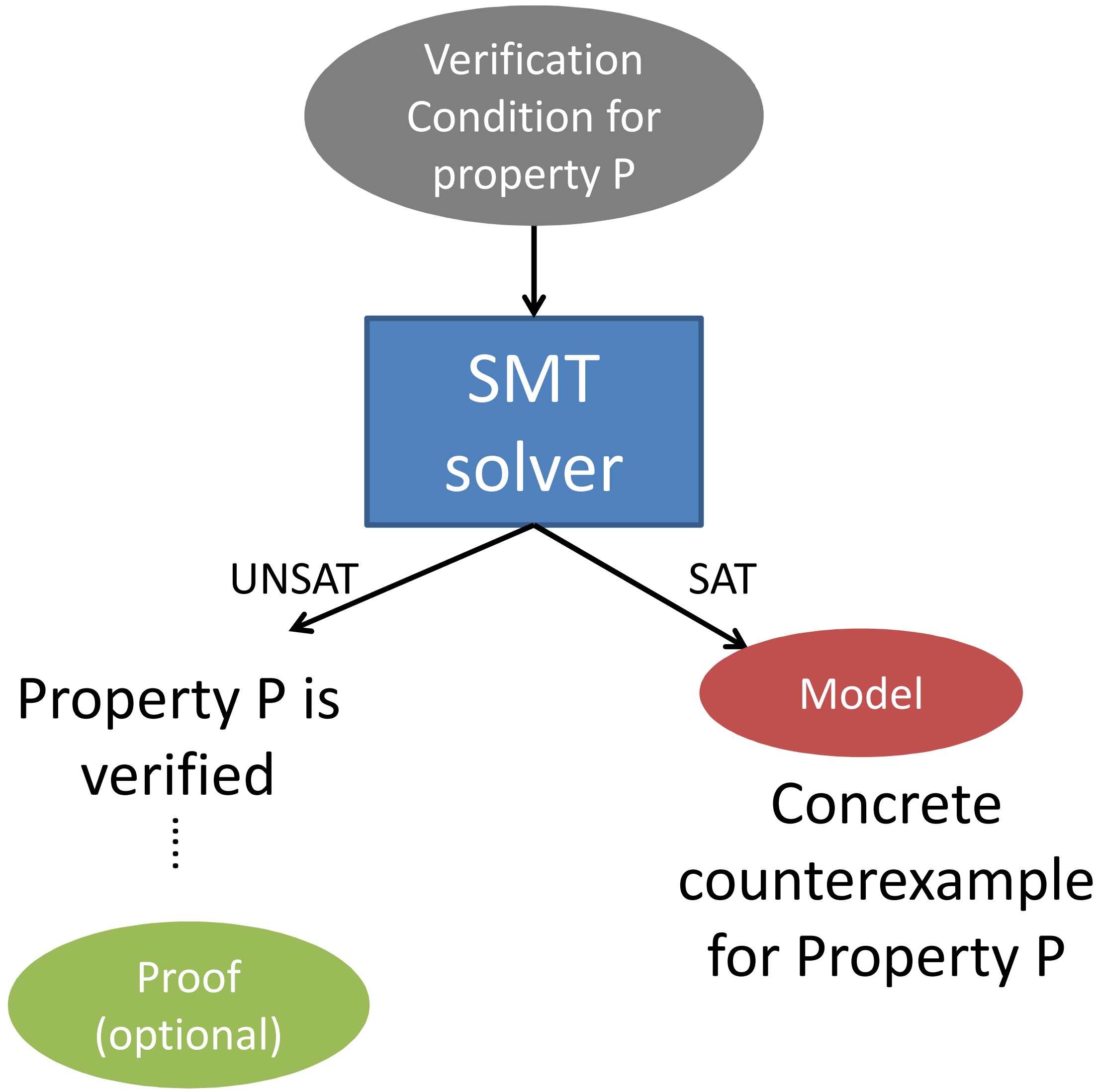
Axioms:

$\forall$x : R. count[x] > 0 $\Rightarrow$ valid[x]
$\forall$x : P. $\neg$ mem(empty, x)
$\forall$x : S, y, z : P. mem(add(x, y), z) $\Rightarrow$ (z = y $\vee$ mem(x, z))
$\forall$x : S, y, z : P. mem(remove(x, y), z) $\Rightarrow$ (z $\neq$ y $\wedge$ mem(x, z))
…

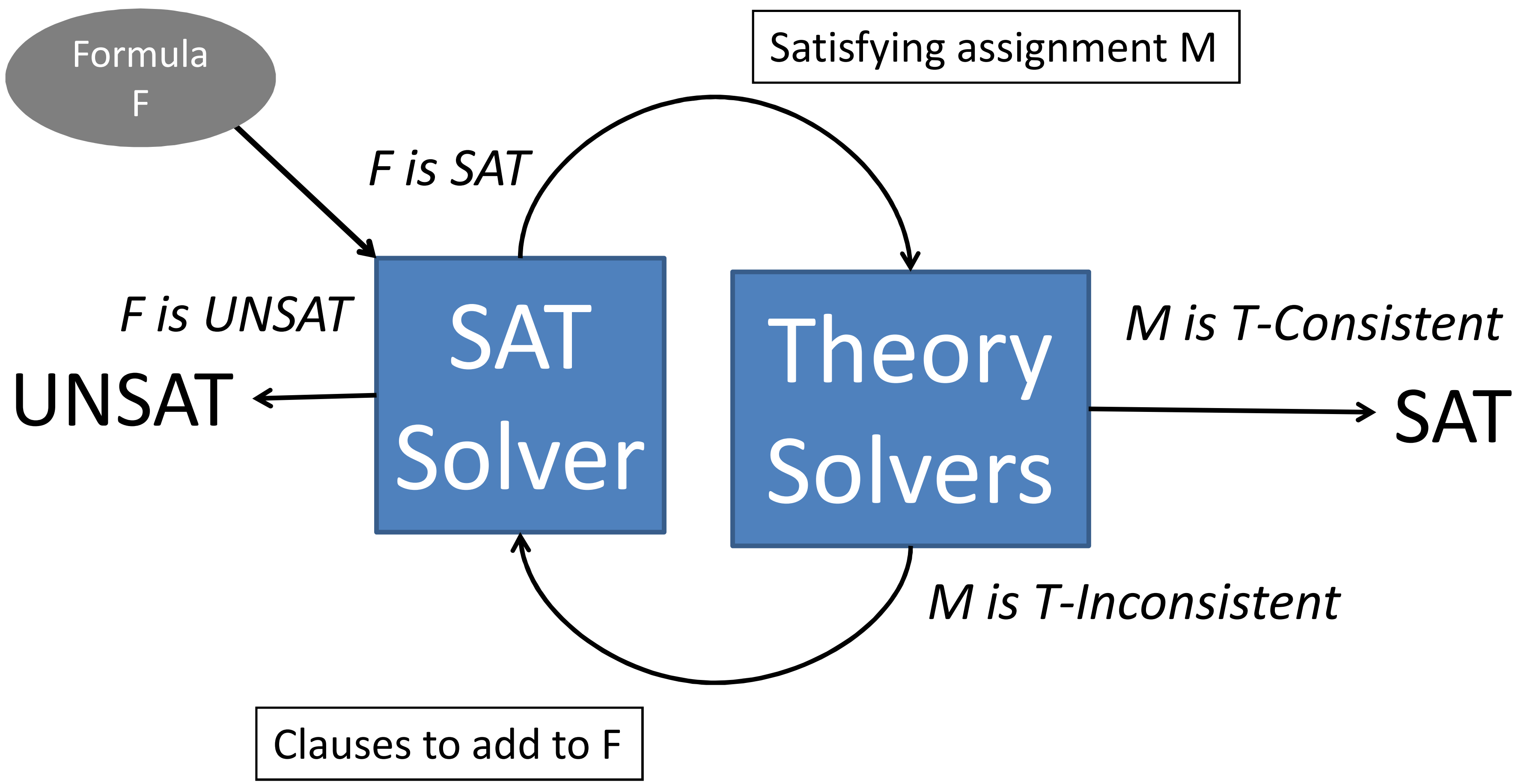
$\neg$ (… $\forall$x. (ref[x] != null => valid[ref[x]]) …)

Property to verify

SMT for Verification Conditions



SMT: DPLL(T) Architecture



Why Quantifiers?

- Quantifiers exist in verification conditions:

Definitions

S, P, R : type
null : R
valid: Array(R, Bool)
count: Array(R, Int)
ref: Array(P, R)
empty : S
mem : (S, P) -> Bool
add : (S, P) -> S

Axioms

$\forall x : R.\ count[x] > 0 \Rightarrow valid[\ x\ ]$
$\forall x : P.\ \neg\ mem(\ empty, x\ )$
$\forall x : S, y, z : P.\ mem(\ add(\ x, y\ ), z\ ) \Rightarrow (\ z = y \vee mem(\ x, z\ )\ )$
$\forall x : S, y, z : P.\ mem(\ remove(\ x, y\ ), z\ ) \Rightarrow (\ z \neq y \wedge mem(\ x, z\ )\ )$
…

$\neg$ (… $\forall$x. (ref[x] != null => valid[ref[x]]) …)

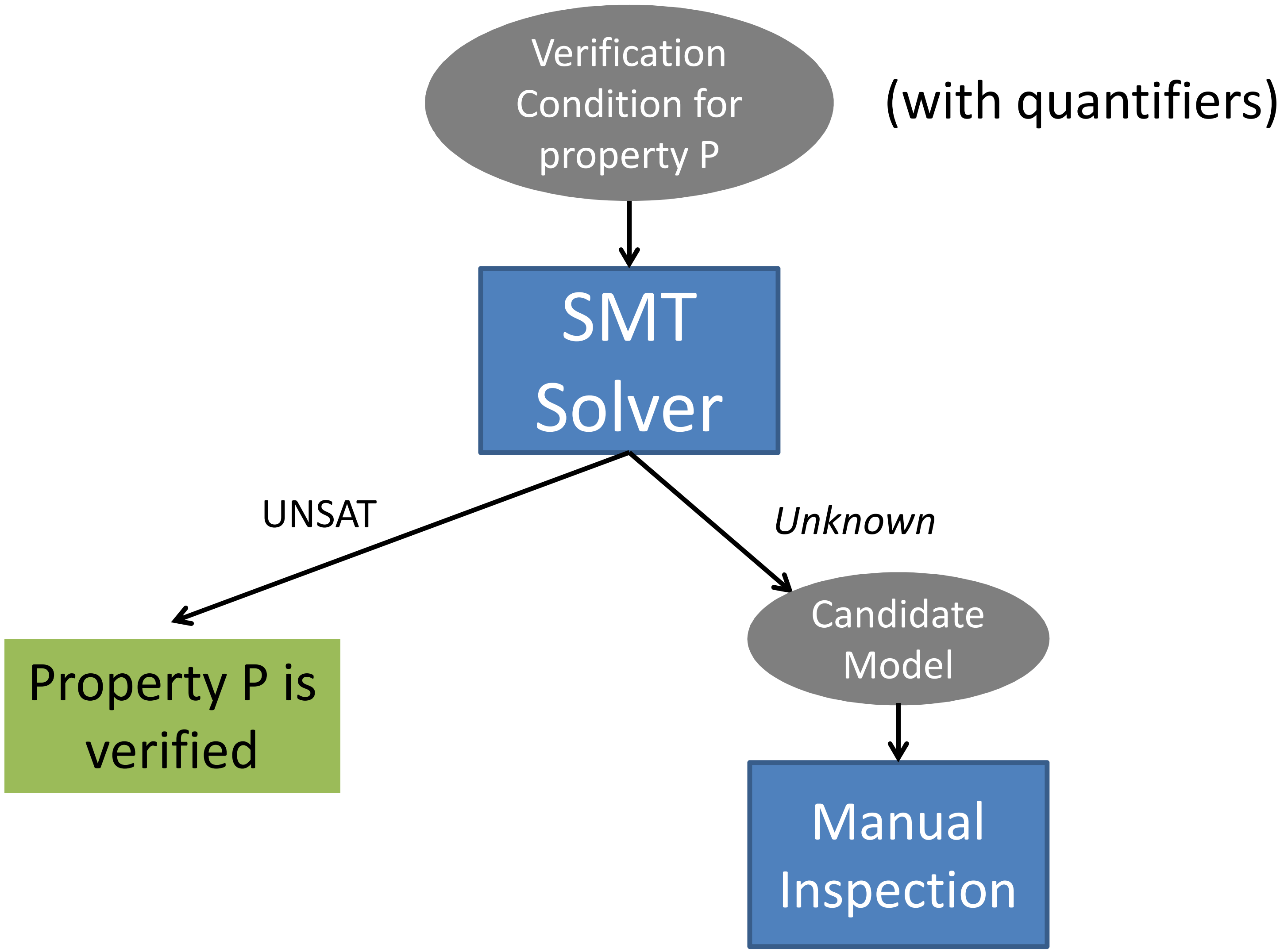
Property to verify

Challenge of Quantifiers in SMT

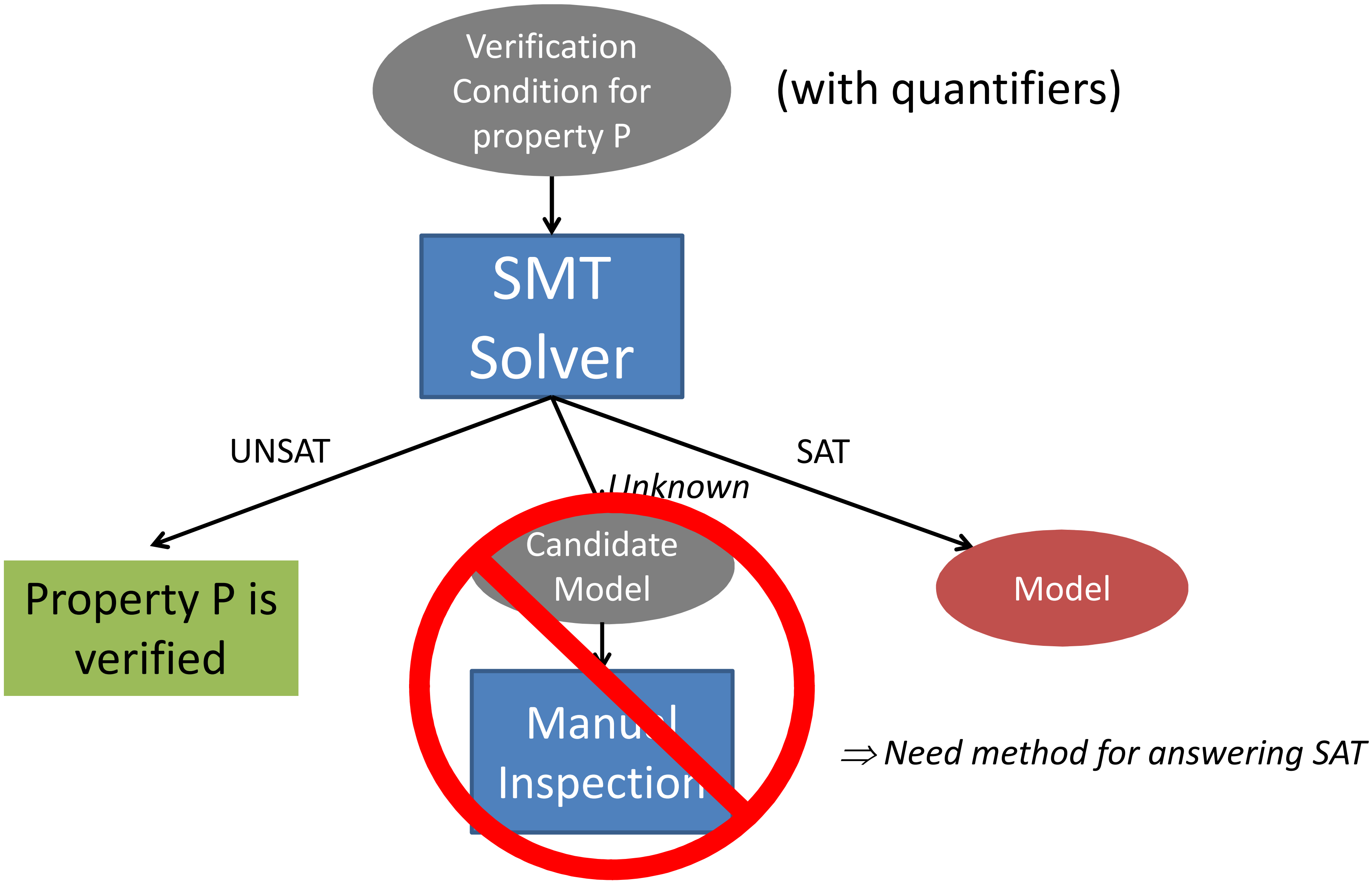
- In general, determining T-consistency of a set of quantified formulas is *undecidable*

- SMT solvers will typically:
  - Add ground instances of quantified formulas
    - i.e. for $\forall$x. F, add lemmas F[t_1/x], F[t_2/x], …
      - If ground conflict exists, answer UNSAT
      - Otherwise, may continue indefinitely
  - Sound but incomplete procedure

Handling Verification Conditions



Handling Verification Conditions



Finite Model Finding

- Method to answer SAT in presence of quantifiers
- Given (G, Q):
  - Set of ground constraints G
  - Set of quantified assertions Q

1. Find a "smallest" model for G
   - Least number of equivalence classes for terms
2. Try *every* instance of Q in the model
   - Feasible if # eq classes we need to consider is *finite*
3. If every instance is true in model, answer SAT

- Consider quantifiers over *uninterpreted sorts*
  - Values, Addresses, Processes, Resources, Sets, …

Finite Model Finding : Example

$$\underbrace{a \neq b,\ b = c,}_{G}\ \underbrace{\forall x.\ f(x) = x}_{Q}$$

1. Smallest model for G, size 2 : $\{ \underline{a} \}, \{ \underline{b}, c \}$
2. Substitute Q with [a/x], [b/x]:
   - $f(a) = a$, $f(b) = b$ added to G
3. Afterwards: $\{ \underline{a}, f(a) \}, \{ \underline{b}, c, f(b) \}$
   - All instances are true

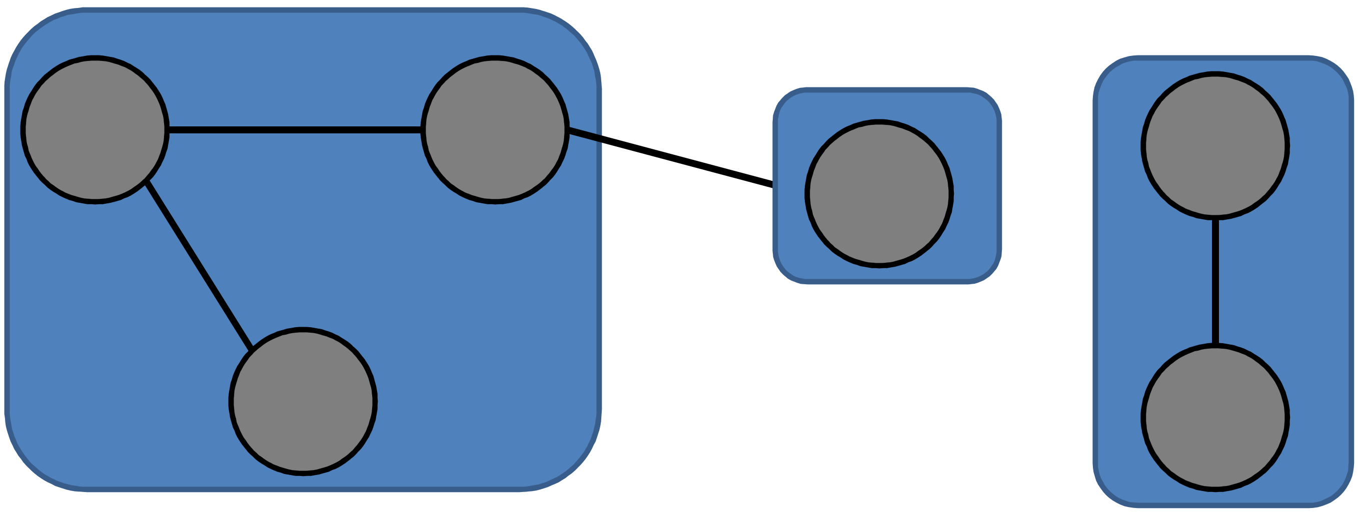
$\Rightarrow$ answer SAT

Finding Small Models

- "Smallest" model for sort S means:
  - Fewest # equivalence classes of sort S
- To find small models:
  - Try to find models of size 1, 2, 3, … etc.
    - Impose *cardinality constraints*
- Requires solver for equality with cardinality constraints

Solver for Eq + Cardinality Constraints

- Maintain disequality graph
  - Nodes are equivalence classes
  - Edges are disequalities
- For cardinality k, interested whether graph is k-colorable

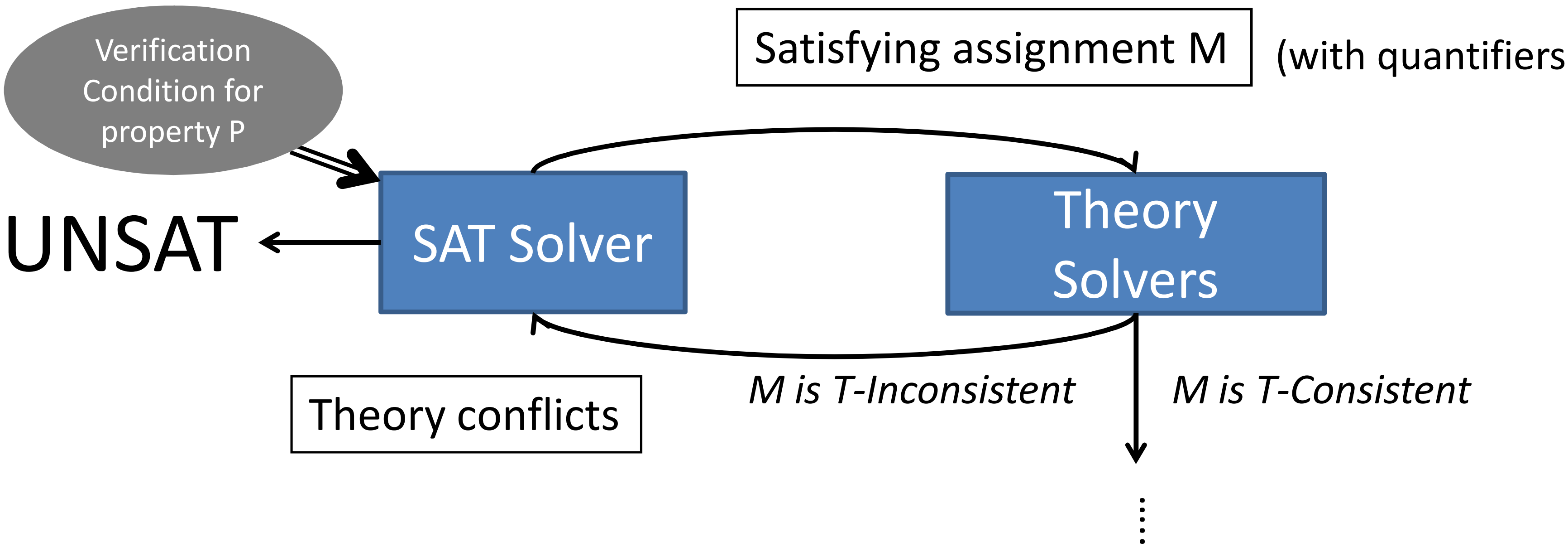


- Partition disequality graph of the solver into *regions* where the edge density is high, so that we:
  - Discover cliques local to regions
  - Suggest relevant terms to identify

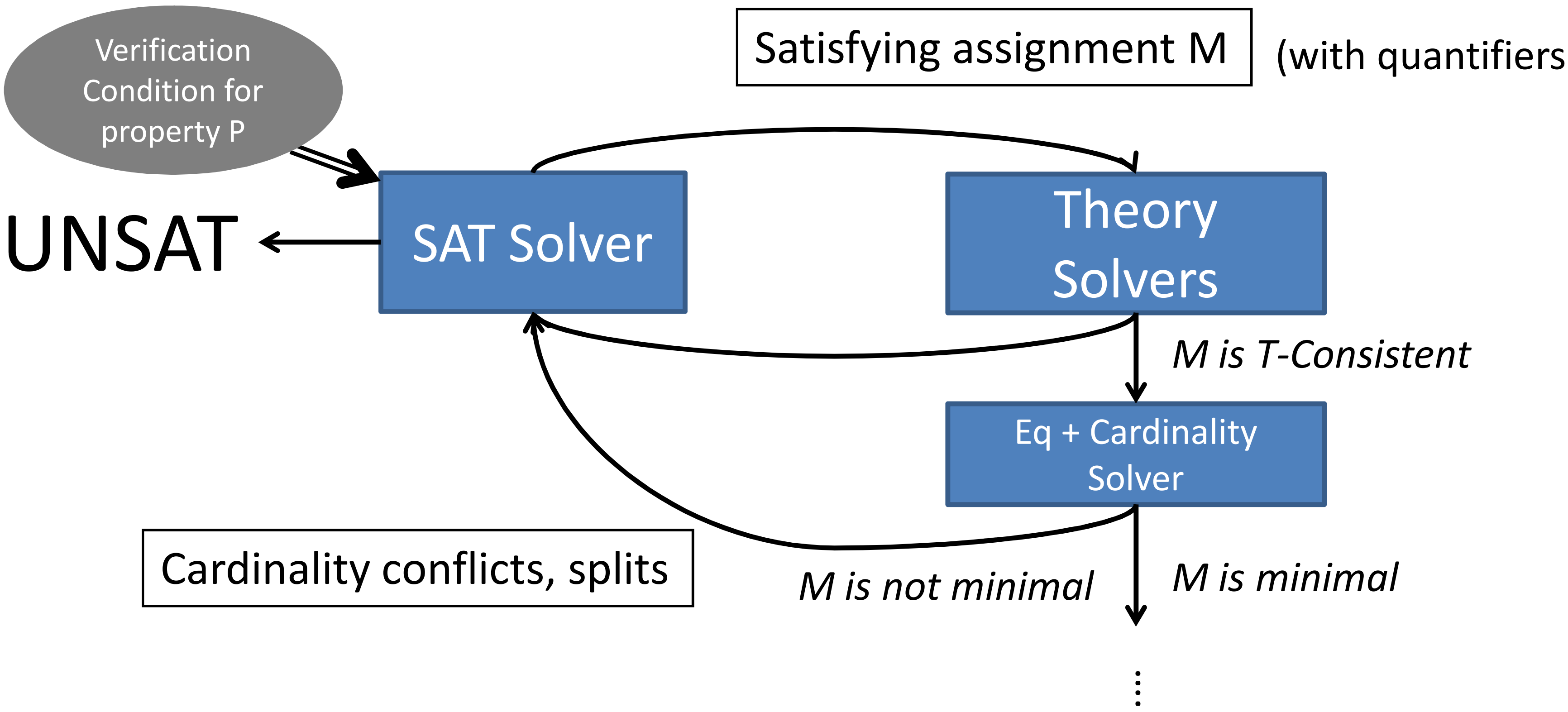
Why Small Models?

- Easier to test against quantifiers
  - Given quantified formula $\forall x_1 \ldots x_n$. F
    - Naively, we require k^n instantiations,
      - where k is the cardinality of sort($x_1 \ldots x_n$)
  - Feasible if either:
    - Both n and k are small
    - We can recognize/eliminate redundant instantiations
      - *Model-Based Quantifier Instantiation* [Ge/deMoura 09]
      - i.e. do not consider instances that are already true

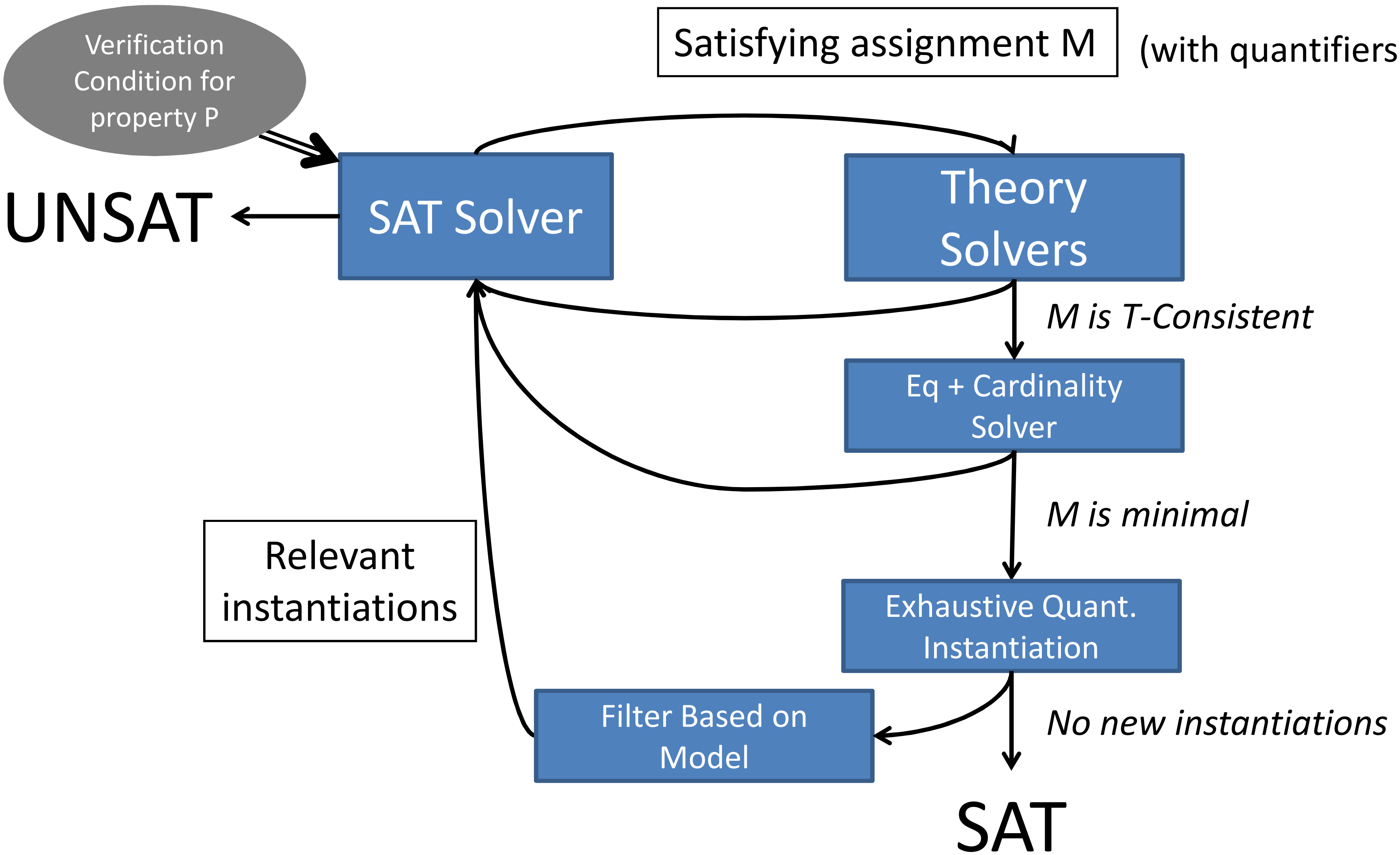
Anatomy of Finite Model Finding



Anatomy of Finite Model Finding



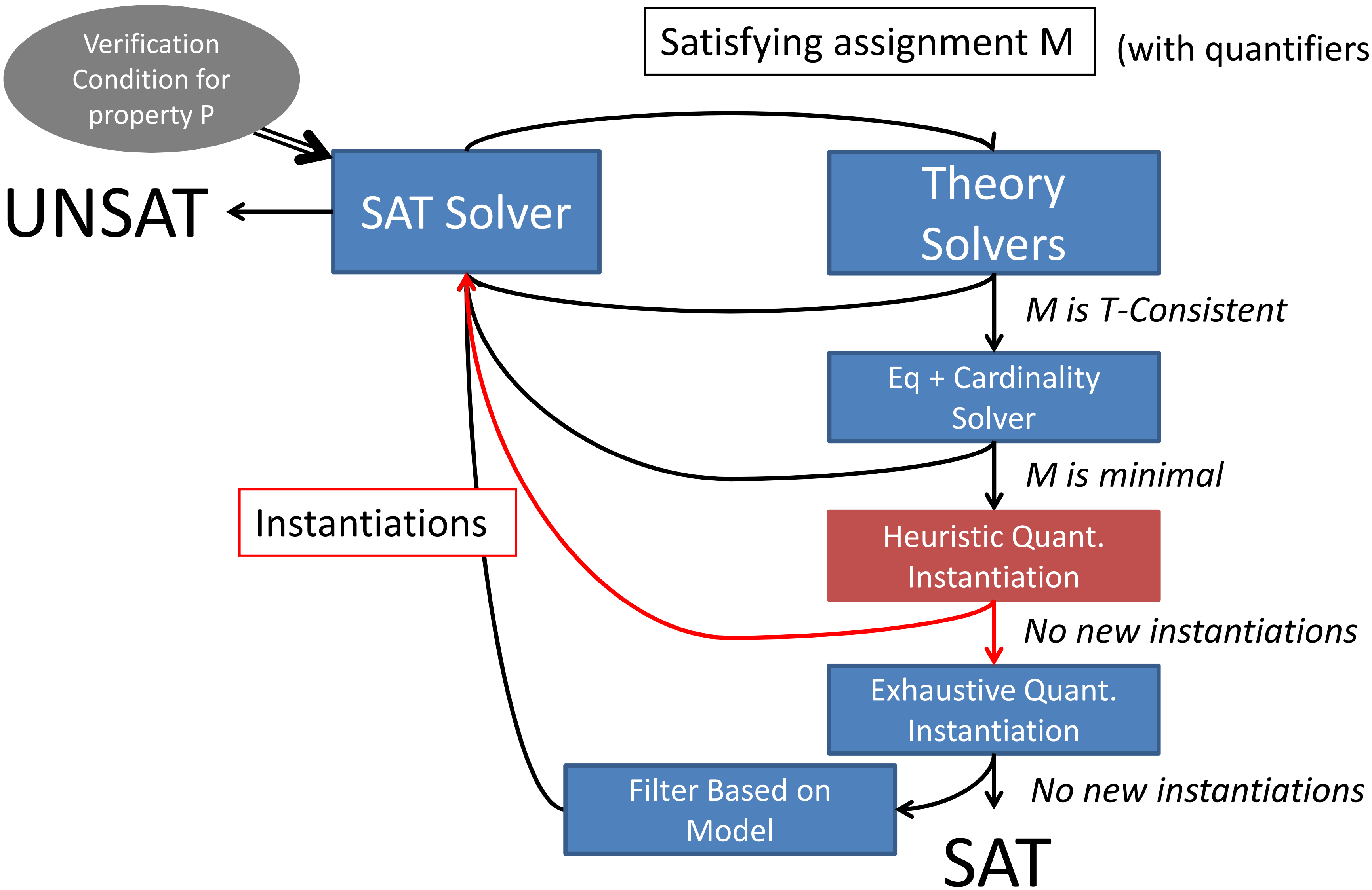
Anatomy of Finite Model Finding



FMF + Heuristic Instantiation

- Idea:
  - First see if instantiations based on heuristics exist
  - If not, resort to exhaustive instantiation
- May lead to:
  - Answering UNSAT more often
    - Discover easy conflicts, if they exist
  - Arriving at model faster
    - Instantiations rule out spurious models

FMF + Heuristic Instantiation



Experimental Results

- Implemented in SMT Solver CVC4
- DVF Benchmarks
  - Taken from real examples of interest to Intel
  - Both SAT/UNSAT benchmarks
    - SAT benchmarks generated by removing necessary pf assumptions
  - Many theories: UF, arithmetic, arrays, datatypes
- TPTP Benchmarks
  - Taken from ATP community
  - Heavily quantified
  - Unsorted logic

Results: DVF

UNSAT	german	refcount	agree	apg	bmk	Total
cvc4	**145**	**40**	600	**304**	**244**	1333
cvc4+fmf	**145**	**40**	**604**	294	236	1319
z3	**145**	**40**	**604**	**304**	**244**	**1337**
	145	40	604	304	244	1337

SAT	german	refcount	agree	apg	bmk	Total
cvc4	2	0	0	0	0	2
cvc4+fmf	**45**	**6**	**62**	**16**	**36**	**165**
z3	**45**	1	0	0	0	46
	45	6	62	19	37	169

- 60 second timeout

Results: TPTP

- 10 second timeout
- 11613 UNSAT benchmarks:
  - z3: **5471** solved
  - cvc4: 4868 solved
  - cvc4+fmf: 2246 solved, but orthogonal
    - 288 solved that cvc4 w/o finite model finding cannot
  - Either cvc4 or cvc4+fmf: 5158 solved
- 1933 SAT benchmarks:
  - z3: 866 solved
  - cvc4+fmf: **920** solved
- Model-Based filtering of instances is essential

Summary

- Finite model finding in CVC4:
  - Finds minimal models for ground constraints
  - Uses exhaustive instantiation to test models
    - Instantiations filtered by model
  - Optionally, uses heuristic instantiation

Conclusions

- Finite Model Finding:
  - Practical approach for SMT + quantifiers
  - Can answer SAT quickly
    - Generate simple counterexamples for DVF
      - Many models in real examples have cardinality 2 or 3
  - Improves coverage in UNSAT cases
    - Increased ability to discharge verification conditions
  - Orthogonal to other approaches

Questions?