SMT Techniques for Fast Predicate Abstraction

Shuvendu K. Lahiri[1], Robert Nieuwenhuis[2], and Albert Oliveras[2]

[1] Microsoft Research, Redmond
[2] Technical University of Catalonia

CAV'06, Seattle

Overview of the talk

- Predicate abstraction
    - Introduction
    - Existing methods
- Satisfiability Modulo Theories
    - Introduction
    - Eager and lazy approach
- SMT for Predicate Abstraction
    - Basic idea
    - All-SAT algorithms
    - Experimental evaluation
    - Incremental refinement
- Conclusions and future work

Predicate abstraction - Overview

- Model checking validates and debugs systems by exploration of their state spaces
- PROBLEM: state-space explosion
  - Hardware and protocols: very large number of states
  - Software: typically infinite-state
- SOLUTION: analyze a finite-state abstraction of the system

PREDICATE ABSTRACTION [Graf and Saïdi, CAV'97]:

- INPUT: a concrete system $\mathcal{C}$ (states + transition relation) and a set of predicates P (properties of the system)
- OUTPUT: finite-state *conservative* abstraction $\mathcal{A}$.
  (e.g. abstraction of state is the evaluation of P on it)

  Conservative: if a property holds in $\mathcal{A}$, a concrete version holds in $\mathcal{C}$

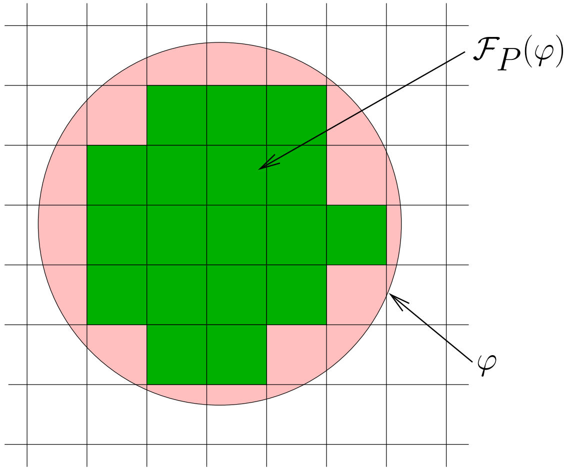
Predicate abstraction - Key operation

PREDICATE ABSTRACTION-KEY OPERATION:

- INPUT:
  - A theory T
  - A formula φ (representing, e.g., a set of concrete states)
  - A set of predicates $P = \{P_1, \ldots, P_n\}$ describing some set of properties of the system state
- OUTPUT: the most precise T-approximation of φ using P

This amounts to compute either

- $\mathcal{F}_P(\varphi)$: the weakest Boolean expression over P that T-implies φ, or



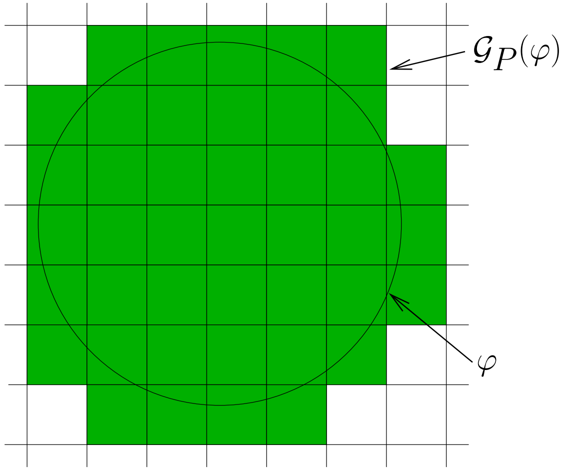
Predicate abstraction - Key operation

PREDICATE ABSTRACTION-KEY OPERATION:

- INPUT:
  - A theory T
  - A formula φ (representing, e.g., a set of concrete states)
  - A set of predicates $P = \{P_1, \ldots, P_n\}$ describing some set of properties of the system state
- OUTPUT: the most precise T-approximation of φ using P

This amounts to compute either

- $\mathcal{F}_P(\varphi)$: the weakest Boolean expression over P that T-implies φ, or
- $\mathcal{G}_P(\varphi)$: the strongest Boolean expression over P T-implied by φ



Predicate abstraction - Example

INPUT: $\varphi \equiv \quad x < y - 2 \quad \vee \quad x > y$

$$P = \{\underbrace{x < 0}_{p_1}, \underbrace{y = 2}_{p_2}, \underbrace{x \neq 4}_{p_3})\}$$

OUTPUT: $\mathcal{F}_P(\varphi)$, the weakest formula over P T-entailing φ, is

$$(p_1 \wedge p_2) \vee (p_2 \wedge \neg p_3)$$

Clearly:

- $x < 0, y = 2 \quad \models_T \quad x < y - 2 \quad \vee \quad x > y$
- $y = 2, x = 4 \quad \models_T \quad x < y - 2 \quad \vee \quad x > y$

But, is it the weakest such formula?

Predicate abstraction - Computation

Some notation:

- A cube is a conjunction of literals of P.
- A minterm is a cube of size $|P|$ with exactly one of P_i or $\neg P_i$.

The computation of $\mathcal{F}_P(\varphi)$ and $\mathcal{G}_P(\varphi)$ is given by:

- $\mathcal{F}_P(\varphi)$ is $\bigvee\{c \mid c$ is a minterm over P and $c \models_T \varphi\}$,
- $\mathcal{G}_P(\varphi)$ is $\neg\mathcal{F}_P(\neg\varphi)$.
- $\mathcal{G}_P(\varphi)$ is $\bigvee\{c \mid c$ is a minterm over P and $c \wedge \varphi$ is T-satisfiable$\}$,

ALGORITHM:

Check, for each minterm c, whether $c \wedge \varphi$ is T-satisfiable.

Predicate abstraction - Existing methods

Three main approaches (in chronological order):

- Check satisfiability of $c \wedge \varphi$ for all minterms c (exponential number of calls):
  - [Saidi and Shankar, CAV'99]: up to 3^n calls
  - [Das et al, CAV'99]: up to 2^{n+1} calls
  - [Flanagan and Qaader, POPL'02]: up to $n \cdot 2^n$ calls
- Reduce the problem to Boolean quantifier elimination (and use SAT-solving techniques):
  - [Lahiri et al, CAV'03]
  - [Clarke et al, FMSD'04]
- Use symbolic decision procedures (symbolic execution of decision procedures) [Lahiri et al, CAV'05]

Overview of the talk

- Predicate abstraction
  - Introduction
  - Existing methods
- Satisfiability Modulo Theories
  - Introduction
  - Eager and lazy approach
- SMT for Predicate Abstraction
  - Basic idea
  - All-SAT algorithms
  - Experimental evaluation
  - Incremental refinement
- Conclusions and future work

Introduction to SMT

- Some problems are more naturally expressed in other logics than propositional logic, e.g:
  - Software verification needs reasoning about equality, arithmetic, data structures, ...
- SMT consists of deciding the satisfiability of a (ground) FO formula with respect to a background theory T
- Example (Equality with Uninterpreted Functions – EUF):

$$g(a)=c \quad \wedge \quad (\ f(g(a)) \neq f(c) \ \vee \ g(a)=d\ ) \quad \wedge \quad c \neq d$$

- Wide range of applications:
  - Predicate abstraction
  - Model checking
  - Equivalence checking
  - Static analysis
  - Scheduling
  - ...

SMT - Eager approach vs lazy approach

EAGER APPROACH:

- Methodology: translate problem into equisatisfiable propositional formula and use off-the-shelf SAT solver [Bryant, Velev, Pnueli, Lahiri, Seshia, Strichman, ...]
- Why "eager"? Search uses all theory information from the beginning
- Tools: UCLID [Lahiri, Seshia and Bryant]

LAZY APPROACH:

- Methodology: integration of a SAT-solver with a theory solver
- Why "lazy"? Theory information used lazily when checking T-consistency of propositional models
- Tools: CVC-Lite, Yices, MathSAT, TSAT+, Barcelogic ...

SMT - Lazy approach example

Consider EUF and

$$\underbrace{g(a)=c}_{1} \quad\wedge\quad (\underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}) \quad\wedge\quad \underbrace{c\neq d}_{\overline{4}}$$

- Send $\{1,\ \overline{2}\vee 3,\ \overline{4}\}$ to SAT solver

SMT - Lazy approach example

Consider EUF and

$$\underbrace{g(a)=c}_{1} \quad \wedge \quad (\ \underbrace{f(g(a))\neq f(c)}_{\overline{2}} \ \vee \ \underbrace{g(a)=d}_{3}\ ) \quad \wedge \quad \underbrace{c\neq d}_{\overline{4}}$$

- Send $\{1,\ \ \overline{2} \vee 3,\ \ \overline{4}\}$ to SAT solver
  - SAT solver returns model $[1,\ \ \overline{2},\ \ \overline{4}]$

SMT - Lazy approach example

Consider EUF and

$$\underbrace{g(a)=c}_{1} \quad \wedge \quad (\ \underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}\ ) \quad \wedge \quad \underbrace{c\neq d}_{\overline{4}}$$

- Send $\{1,\ \ \overline{2}\vee 3,\ \ \overline{4}\}$ to SAT solver
  - SAT solver returns model $[1,\ \ \overline{2},\ \ \overline{4}]$
  - Theory solver says T-inconsistent

SMT - Lazy approach example

Consider EUF and

$$\underbrace{g(a)=c}_{1} \quad \wedge \quad (\ \underbrace{f(g(a))\neq f(c)}_{\overline{2}} \ \vee\ \underbrace{g(a)=d}_{3}\ ) \quad \wedge \quad \underbrace{c\neq d}_{\overline{4}}$$

- Send $\{1,\ \ \overline{2}\vee 3,\ \ \overline{4}\}$ to SAT solver
  - SAT solver returns model $[1,\ \ \overline{2},\ \ \overline{4}]$
  - Theory solver says T-inconsistent
- Send $\{1,\ \ \overline{2}\vee 3,\ \ \overline{4},\ \ \overline{1}\vee 2\vee 4\}$ to SAT solver

SMT - Lazy approach example

Consider EUF and

$$\underbrace{g(a)=c}_{1} \quad \wedge \quad (\ \underbrace{f(g(a))\neq f(c)}_{\overline{2}} \ \vee \ \underbrace{g(a)=d}_{3}\ ) \quad \wedge \quad \underbrace{c\neq d}_{\overline{4}}$$

- Send $\{1,\ \ \overline{2}\vee 3,\ \ \overline{4}\}$ to SAT solver
  - SAT solver returns model $[1,\ \ \overline{2},\ \ \overline{4}]$
  - Theory solver says T-inconsistent
- Send $\{1,\ \ \overline{2}\vee 3,\ \ \overline{4},\ \ \overline{1}\vee 2\vee 4\}$ to SAT solver
  - SAT solver returns model $[1, 2, 3, \overline{4}]$

SMT - Lazy approach example

Consider EUF and

$$\underbrace{g(a)=c}_{1} \quad \wedge \quad (\ \underbrace{f(g(a))\neq f(c)}_{\overline{2}} \ \vee\ \underbrace{g(a)=d}_{3}\ ) \quad \wedge \quad \underbrace{c\neq d}_{\overline{4}}$$

- Send $\{1,\ \ \overline{2}\vee 3,\ \ \overline{4}\}$ to SAT solver
  - SAT solver returns model $[1,\ \ \overline{2},\ \ \overline{4}]$
  - Theory solver says T-inconsistent
- Send $\{1,\ \ \overline{2}\vee 3,\ \ \overline{4},\ \ \overline{1}\vee 2\vee 4\}$ to SAT solver
  - SAT solver returns model $[1, 2, 3, \overline{4}]$
  - Theory solver says T-inconsistent

SMT - Lazy approach example

Consider EUF and

$$\underbrace{g(a)=c}_{1} \quad \wedge \quad (\underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}) \quad \wedge \quad \underbrace{c\neq d}_{\overline{4}}$$

- Send $\{1,\ \overline{2}\vee 3,\ \overline{4}\}$ to SAT solver
  - SAT solver returns model $[1,\ \overline{2},\ \overline{4}]$
  - Theory solver says T-inconsistent
- Send $\{1,\ \overline{2}\vee 3,\ \overline{4},\ \overline{1}\vee 2\vee 4\}$ to SAT solver
  - SAT solver returns model $[1, 2, 3, \overline{4}]$
  - Theory solver says T-inconsistent
- Send $\{1,\ \overline{2}\vee 3,\ \overline{4},\ \overline{1}\vee 2\vee 4,\ \overline{1}\vee\overline{2}\vee\overline{3}\vee 4\}$ to SAT solver

SMT - Lazy approach example

Consider EUF and

$$\underbrace{g(a)=c}_{1} \quad \wedge \quad (\underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}) \quad \wedge \quad \underbrace{c\neq d}_{\overline{4}}$$

- Send $\{1,\ \overline{2}\vee 3,\ \overline{4}\}$ to SAT solver
  - SAT solver returns model $[1,\ \overline{2},\ \overline{4}]$
  - Theory solver says T-inconsistent
- Send $\{1,\ \overline{2}\vee 3,\ \overline{4},\ \overline{1}\vee 2\vee 4\}$ to SAT solver
  - SAT solver returns model $[1, 2, 3, \overline{4}]$
  - Theory solver says T-inconsistent
- Send $\{1,\ \overline{2}\vee 3,\ \overline{4},\ \overline{1}\vee 2\vee 4,\ \overline{1}\vee\overline{2}\vee\overline{3}\vee 4\}$ to SAT solver
  - SAT solver detects it UNSATISFIABLE

SMT - Lazy approach optimizations

Several optimizations for enhancing efficiency:

- Check T-consistency only of full propositional models

SMT - Lazy approach optimizations

Several optimizations for enhancing efficiency:

- ~~Check T-consistency only of full propositional models~~
- Check T-consistency of partial assignment while being built

SMT - Lazy approach optimizations

Several optimizations for enhancing efficiency:

- ~~Check T-consistency only of full propositional models~~
- Check T-consistency of partial assignment while being built

- Given a T-inconsistent assignment M, add $\neg M$ as a clause

SMT - Lazy approach optimizations

Several optimizations for enhancing efficiency:

- ~~Check T-consistency only of full propositional models~~
- Check T-consistency of partial assignment while being built

- ~~Given a T-inconsistent assignment M, add $\neg M$ as a clause~~
- Given a T-inconsistent assignment M, identify a T-inconsistent subset $M_0 \subseteq M$ and add $\neg M_0$ as a clause

SMT - Lazy approach optimizations

Several optimizations for enhancing efficiency:

- ~~Check T-consistency only of full propositional models~~
- Check T-consistency of partial assignment while being built

- ~~Given a T-inconsistent assignment M, add $\neg M$ as a clause~~
- Given a T-inconsistent assignment M, identify a T-inconsistent subset $M_0 \subseteq M$ and add $\neg M_0$ as a clause

- Upon a T-inconsistency, add clause and restart

SMT - Lazy approach optimizations

Several optimizations for enhancing efficiency:

- ~~Check T-consistency only of full propositional models~~
- Check T-consistency of partial assignment while being built

- ~~Given a T-inconsistent assignment M, add $\neg M$ as a clause~~
- Given a T-inconsistent assignment M, identify a T-inconsistent subset $M_0 \subseteq M$ and add $\neg M_0$ as a clause

- ~~Upon a T-inconsistency, add clause and restart~~
- Upon a T-inconsistency, use the conflicting clause $\neg M_0$ to backjump to some point where the assignment was still T-consistent, as in SAT-solvers.

Overview of the talk

- Predicate abstraction
  - Introduction
  - Existing methods
- Satisfiability Modulo Theories
  - Introduction
  - Eager and lazy approach
- SMT for Predicate Abstraction
  - Basic idea
  - All-SAT algorithms
  - Experimental evaluation
  - Incremental refinement
- Conclusions and future work

SMT for Predicate Abstraction

INPUT: a formula φ, a set of predicates P and a theory T
OUTPUT: $\mathcal{G}_P(\varphi) \equiv \bigvee\{c \mid c$ is a minterm over P and $c \wedge \varphi$ is T-sat$\}$

IDEA:

- introduce n fresh propositional variables $B = \{b_1, \ldots, b_n\}$
- consider the formula $\psi \equiv \varphi \wedge_{i=1}^{n} (b_i \leftrightarrow P_i)$
- given a model M of ψ, project it onto B, i.e., collect the conjunction of all B-literals in M and then replace each b_i by P_i. This gives a minterm c in $\mathcal{G}_P(\varphi)$
- repeat the previous step for all models M

MISSING POINT:

- need All-SAT mechanism to compute all models M

Computation of all models of ψ

FIRST IDEA (black-box approach):

```
while $\psi$ is $T$-satisfiable do
    - Let the SMT-solver find a model $M$ of $\psi$

    - $\psi := \psi \wedge \neg M$
end while
```

Computation of all models of ψ

FIRST IDEA (black-box approach):

```
while $\psi$ is $T$-satisfiable do
    - Let the SMT-solver find a model $M$ of $\psi$
    - $M :=$ projection of $M$ onto $B$
    - $\psi := \psi \wedge \neg M$
end while
```

Computation of all models of ψ

FIRST IDEA (black-box approach):

```
while $\psi$ is $T$-satisfiable do
    - Let the SMT-solver find a model $M$ of $\psi$
    - $M :=$ projection of $M$ onto $B$
    - $\psi := \psi \wedge \neg M$
end while
```

- Termination: each loops precludes a minterm, and there are only finitely many
- Calls to the SMT-solver are independent:
  - \+ any off-the-shelf SMT-solver can be used
  - \- no computations are reused between calls
- Size of ψ may grow exponentially: 2^n minterms to preclude (however, note that typically n not larger than 30)

Computations of all models of ψ (II)

SECOND IDEA (naive approach):

- After adding $\neg M$ to the formula, restart the SMT-solver but reusing all generated lemmas

Computations of all models of ψ (II)

SECOND IDEA (naive approach):

- After adding $\neg M$ to the formula, restart the SMT-solver but reusing all generated lemmas

THIRD IDEA (refined approach):

- Instead of adding $\neg M$ to the formula do:
  1. Consider $\neg M$ as a conflicting clause
  2. Apply conflict analysis mechanism to $\neg M$ generating a backjump clause (add it if wanted)
  3. Backjump and continue the search for a model
- Termination: more information at lower decision levels
- ψ does not grow, but models may be enumerated more than once

Experimental evaluation

- All three approaches were implemented on top of BarcelogicTools SMT-Solver (BCLT)
- The BDD package CUDD[Somenzi] was used to collect all models and extract a compact representation
- Benchmarks from different sources, but all of them are EUF + Difference Logic, where atoms are, for example:

$$f(g(a), b) - c \leq 4$$

- Nelson-Oppen was not used, we used Ackermann's reduction instead to convert them into Difference Logic:
  - find two terms $f(a_1, \ldots, a_n)$ and $f(b_1, \ldots, b_n)$
  - replace them with c_a and c_b
  - add the clause $a_1 = b_1 \wedge \ldots \wedge a_n = b_n \;\; \rightarrow \;\; c_a = c_b$

Experimental results

- Microsoft SLAM project: Windows device drivers verification
- Initially, theorem prover ZAP [Ball et al, CAV'04] was used for predicate abstraction
- Specialized Symbolic Decision Procedures (SDP) [Lahiri et al, CAV'05] obtained 100x speedup factor
- The biggest availabe set of benchmarks (around 700 queries) processed by SDP in around 700 seconds
- BarcelogicTools only took 5 seconds, another 100x speedup

Experimental results (II)

Hardware and protocol verification benchmarks used in [Lahiri and Bryant, CAV'04]:

Benchmark family	# preds	UCLID	BCLT	
		time (secs.)	time (secs.)	speedup
UCLID Suite:				
aodv	21	657	4.6	143x
bakery	32	245	11	22x
BRP	22	3.5	0.1	35x
cache_ibm	16	34	1.3	26x
cache_bounded	26	1119	23	49x
DLX	23	335	13	26x
OOO	25	921	36	26x

Experimental results (II)

Benchmarks from the verification of programs manipulating linked lists (Qaader and Lahiri, POPL'06):

Benchmark family	**# preds.**	**UCLID**	**BCLT**	
		time (secs.)	**time (secs.)**	**speedup**
Rec. Data Struct.:				
reverse_acyclic	16	20	0.6	33x
set_union	24	22	0.7	31x
simple_cyclic	15	3.7	0.11	34x
sorted_int	21	765	19	40x

Analysis of results - different settings

Benchmark family	**# minterms**	**BCLT (time in secs.)**			**# cubes in adv.**
		black-box	**naive**	**refined**	
UCLID Suite:					
aodv	2916	24	11	4.6	458
bakery	426	19	13	11	294
BRP	30	0.12	0.13	0.1	24
cache_ibm	326	2.3	2	1.3	123
cache_bounded	2238	63	31	23	1022
DLX	30808	242	63	13	2704
OOO	10728	176	57	36	242

Incremental refinement

- Computing $\mathcal{G}_P(\varphi)$ might sometimes be too expensive
- In those cases, a formula implied by φ might be enough
- We have proposed a way to compute a sequence of approximations $\{\mathcal{G}_P^{k_i}(\varphi)\}_{i=1}^m$ such that:
  - Each approximation is more precise than the previous one
  - The last approximation is $\mathcal{G}_P(\varphi)$
  - The sequence can be computed incrementally
- Basically, if $restr(c, k)$ is a subcube of c of size k we have that $\mathcal{G}_P^{k_i}(\varphi) \equiv \bigvee\{restr(c, k_i) \mid c \text{ is a mint. over } P \text{ and } c \wedge \varphi \text{ is } T\text{-sat}\}$
- However, refinement is non-standard (not counter-example-driven)

Incremental refinement - Evaluation

Benchmark family	**#preds.**	Time in seconds **only** $\mathcal{G}_P(\varphi)$	**All sequence in steps of** **step of 1**	 **step of 2**	 **step of 5**
UCLID Suite:					
aodv	21	4.6	15	10	7.2
bakery	32	11	28	21	16
BRP	22	0.1	1.1	0.6	0.3
cache_ibm	16	1.3	3	2.2	1.7
cache_bounded	26	23	71	51	40
DLX	23	13	37	26	18
OOO	25	36	67	50	43

- Step of 2 means computing $\mathcal{G}_P^2(\varphi), \mathcal{G}_P^4(\varphi), \ldots, \mathcal{G}_P^n(\varphi) \equiv \mathcal{G}_P(\varphi)$

Overview of the talk

- Predicate abstraction
  - Introduction
  - Existing methods
- Satisfiability Modulo Theories
  - Introduction
  - Eager and lazy approach
- SMT for Predicate Abstraction
  - Basic idea
  - All-SAT algorithms
  - Experimental evaluation
  - Incremental refinement
- Conclusions and future work

Conclusions and future work

CONCLUSIONS:

- SMT-based predicate abstraction engines can be very efficient
- Very small implementation effort

FUTURE WORK:

- Generation of partial models
- Evaluate practicality of incremental refinement scheme
- Develop refinement schemes over a monotonically growing set of predicates