SMT Techniques for Fast Predicate Abstraction

Shuvendu K. Lahiri\textsuperscript{1}, Robert Nieuwenhuis\textsuperscript{2}, and Albert Oliveras\textsuperscript{2}

\textsuperscript{1} Microsoft Research, Redmond
\textsuperscript{2} Technical University of Catalonia
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 $C$ (states + transition relation) and a set of predicates $P$ (properties of the system)

- **OUTPUT:** finite-state conservative abstraction $A$.
  (e.g. abstraction of state is the evaluation of $P$ on it)

  Conservative: if a property holds in $A$, a concrete version holds in $C$
**Predicate abstraction - Key operation**

**PREDICATE ABSTRACTION - KEY OPERATION:**

- **INPUT:**
  - A theory $T$
  - A formula $\varphi$ (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 $\varphi$ using $P$

This amounts to compute either

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

**PREDICATE ABSTRACTION - KEY OPERATION:**

- **INPUT:**
  - A theory $T$
  - A formula $\varphi$ (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 $\varphi$ using $P$

This amounts to compute either

- $\varphi_P(\varphi)$: the **weakest** Boolean expression over $P$ that $T$-implies $\varphi$, or

- $\varphi_P(\varphi)$: the **strongest** Boolean expression over $P$ $T$-implied by $\varphi$
Predicate abstraction - Example

INPUT: \( \varphi \equiv x < y - 2 \lor x > y \)

\[ P = \{ x < 0, y = 2, x \neq 4 \} \]

\( p_1 \quad p_2 \quad p_3 \)

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

\[ (p_1 \land p_2) \lor (p_2 \land \neg p_3) \]

Clearly:

- \( x < 0, y = 2 \) \( \models_T x < y - 2 \lor x > y \)
- \( y = 2, x = 4 \) \( \models_T x < y - 2 \lor x > y \)

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 $F_P(\varphi)$ and $G_P(\varphi)$ is given by:

- $F_P(\varphi)$ is $\bigvee\{c \mid c$ is a minterm over $P$ and $c \models_T \varphi\}$,
- $G_P(\varphi)$ is $\neg F_P(\neg \varphi)$,
- $G_P(\varphi)$ is $\bigvee\{c \mid c$ is a minterm over $P$ and $c \land \varphi$ is $T$-satisfiable$\}$,

**ALGORITHM:**

Check, for each minterm $c$, whether $c \land \varphi$ is $T$-satisfiable.
Predicate abstraction - Existing methods

Three main approaches (in chronological order):

- **Check satisfiability** of $c \land \varphi$ 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

CAV’06, Seattle
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):

$$g(a) = c \land (f(g(a)) \neq f(c) \lor g(a) = d) \land c \neq d$$

Wide range of applications:

- Predicate abstraction
- Model checking
- Equivalence checking
- Static analysis
- Scheduling
- ...

CAV’06, Seattle
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 ...
Consider EUF and

\[
g(a) = c \land \left( f(g(a)) \neq f(c) \lor g(a) = d \right) \land c \neq d
\]

Send \{1, \overline{2} \lor 3, \overline{4}\} to SAT solver
Consider $\text{EUF}$ and

\[
\begin{align*}
  g(a) &= c \
  1 \\
  \quad \land \quad ( f(g(a)) &\neq f(c) \quad \lor \quad g(a) = d ) \quad \land \quad c \neq d \\
  2 \\
  3 \\
  4
\end{align*}
\]

- Send $\{1, \overline{2} \lor 3, \overline{4}\}$ to SAT solver
- SAT solver returns model $[1, \overline{2}, \overline{4}]$
Consider EUF and

\[
g(a) = c \quad \land \quad \left( f(g(a)) \neq f(c) \quad \lor \quad g(a) = d \right) \quad \land \quad c \neq d
\]

1 \quad \quad 2 \quad \quad 3 \quad \quad 4

- Send \{1, \overline{2} \lor 3, \overline{4}\} to SAT solver
- SAT solver returns model [1, \overline{2}, \overline{4}]
- Theory solver says T-inconsistent
Consider $\text{EUF}$ and

\[
g(a) = c \quad \land \quad \left( f(g(a)) \neq f(c) \lor g(a) = d \right) \quad \land \quad c \neq d
\]

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

- Send \( \{1, \overline{2} \lor 3, \overline{4}, \overline{1} \lor 2 \lor 4\} \) to SAT solver
Consider EUF and

$$g(a) = c \quad \land \quad ( f(g(a)) \neq f(c) \quad \lor \quad g(a) = d ) \quad \land \quad c \neq d$$

- Send \{1, 2 \lor 3, 4\} to SAT solver
  - SAT solver returns model \{1, 2, 4\}
  - Theory solver says T-inconsistent

- Send \{1, 2 \lor 3, 4, 1 \lor 2 \lor 4\} to SAT solver
  - SAT solver returns model \{1, 2, 3, 4\}
Consider EUF and

\[
g(a) = c \quad \land \quad \left( f(g(a)) \neq f(c) \lor g(a) = d \right) \quad \land \quad c \neq d
\]

1

2

3

4

- Send \{1, \ 2 \lor 3, \ 4\} to SAT solver
- SAT solver returns model [1, \ 2, \ 4]
- Theory solver says \(T\)-inconsistent

- Send \{1, \ 2 \lor 3, \ 4, \ 1 \lor 2 \lor 4\} to SAT solver
- SAT solver returns model [1, 2, 3, \ 4]
- Theory solver says \(T\)-inconsistent
Consider EUF and

\[
\begin{align*}
g(a) &= c \quad \land \quad (f(g(a)) \neq f(c) \quad \lor \quad g(a) = d) \\& \quad \land \quad c \neq d
\end{align*}
\]

1 2 3 4

Send \( \{1, \overline{2} \lor 3, \overline{4}\} \) to SAT solver

- SAT solver returns model \( [1, \overline{2}, \overline{4}] \)
- Theory solver says \( T \)-inconsistent

Send \( \{1, \overline{2} \lor 3, \overline{4}, \overline{1} \lor 2 \lor 4\} \) to SAT solver

- SAT solver returns model \( [1, 2, 3, \overline{4}] \)
- Theory solver says \( T \)-inconsistent

Send \( \{1, \overline{2} \lor 3, \overline{4}, \overline{1} \lor 2 \lor 4, \overline{1} \lor \overline{2} \lor \overline{3} \lor 4\} \) to SAT solver
Consider **EUF** and

\[
g(a) = c \quad \land \quad (f(g(a)) \neq f(c) \lor g(a) = d) \quad \land \quad c \neq d
\]

Send \(\{1, \bar{2} \lor 3, \bar{4}\}\) to **SAT** solver

- **SAT** solver returns model \([1, \bar{2}, \bar{4}]\)
- **Theory** solver says **T-inconsistent**

Send \(\{1, \bar{2} \lor 3, \bar{4}, \bar{1} \lor 2 \lor 4\}\) to **SAT** solver

- **SAT** solver returns model \([1, 2, 3, \bar{4}]\)
- **Theory** solver says **T-inconsistent**

Send \(\{1, \bar{2} \lor 3, \bar{4}, \bar{1} \lor 2 \lor 4, \bar{1} \lor \bar{2} \lor 3 \lor 4\}\) to **SAT** solver

- **SAT** solver detects it **UNSATISFIABLE**
Several optimizations for enhancing efficiency:

- Check $T$-consistency only of full propositional models
Several optimizations for enhancing efficiency:

- Check $T$-consistency only of full propositional models.
- Check $T$-consistency of *partial* assignment while being built.
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
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
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.
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

CAV’06, Seattle
SMT for Predicate Abstraction

**INPUT:** a formula $\varphi$, a set of predicates $P$ and a theory $T$

**OUTPUT:** $G_P(\varphi) \equiv \bigvee \{c \mid c \text{ is a minterm over } P \text{ and } c \land \varphi \text{ is } T\text{-sat}\}$

**IDEA:**

- introduce $n$ fresh propositional variables $B = \{b_1, \ldots, b_n\}$
- consider the formula $\psi \equiv \varphi \land \bigwedge_{i=1}^{n} (b_i \leftrightarrow P_i)$
- given a model $M$ of $\psi$, 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 $G_P(\varphi)$
- repeat the previous step for all models $M$

**MISSING POINT:**

- need All-SAT mechanism to compute all models $M$
FIRST IDEA (black-box approach):

while $\psi$ is $T$-satisfiable do

- Let the SMT-solver find a model $M$ of $\psi$

- $\psi := \psi \land \neg M$

end while
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 \( B \)
- \( \psi := \psi \land \neg M \)

end while
FIRST IDEA (black-box approach):

while $\psi$ is $T$-satisfiable do
  - Let the SMT-solver find a model $M$ of $\psi$
  - $M := \text{projection of } M \text{ onto } B$
  - $\psi := \psi \land \neg M$
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 $\psi$ may grow exponentially: $2^n$ minterms to preclude
(however, note that typically $n$ not larger than 30)
SECOND IDEA (naive approach):

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

\( \psi \) 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:

\[ f \left( g(a), b \right) - c \leq 4 \]

- 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 \( a_1 = b_1 \land \ldots \land a_n = b_n \rightarrow c_a = c_b \)
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 available 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]:

<table>
<thead>
<tr>
<th>Benchmark family</th>
<th># preds</th>
<th>UCLID time (secs.)</th>
<th>BCLT time (secs.)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCLID Suite:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aodv</td>
<td>21</td>
<td>657</td>
<td>4.6</td>
<td>143x</td>
</tr>
<tr>
<td>bakery</td>
<td>32</td>
<td>245</td>
<td>11</td>
<td>22x</td>
</tr>
<tr>
<td>BRP</td>
<td>22</td>
<td>3.5</td>
<td>0.1</td>
<td>35x</td>
</tr>
<tr>
<td>cache_ibm</td>
<td>16</td>
<td>34</td>
<td>1.3</td>
<td>26x</td>
</tr>
<tr>
<td>cache_bounded</td>
<td>26</td>
<td>1119</td>
<td>23</td>
<td>49x</td>
</tr>
<tr>
<td>DLX</td>
<td>23</td>
<td>335</td>
<td>13</td>
<td>26x</td>
</tr>
<tr>
<td>OOO</td>
<td>25</td>
<td>921</td>
<td>36</td>
<td>26x</td>
</tr>
</tbody>
</table>
Benchmarks from the verification of programs manipulating linked lists (Qaader and Lahiri, POPL’06):

<table>
<thead>
<tr>
<th>Benchmark family</th>
<th># preds.</th>
<th>UCLID time (secs.)</th>
<th>BCLT time (secs.)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>reverse_acyclic</td>
<td>16</td>
<td>20</td>
<td>0.6</td>
<td>33x</td>
</tr>
<tr>
<td>set_union</td>
<td>24</td>
<td>22</td>
<td>0.7</td>
<td>31x</td>
</tr>
<tr>
<td>simple_cyclic</td>
<td>15</td>
<td>3.7</td>
<td>0.11</td>
<td>34x</td>
</tr>
<tr>
<td>sorted_int</td>
<td>21</td>
<td>765</td>
<td>19</td>
<td>40x</td>
</tr>
</tbody>
</table>
## Analysis of results - different settings

<table>
<thead>
<tr>
<th>Benchmark family</th>
<th># minterms</th>
<th>BCLT (time in secs.)</th>
<th># cubes in adv.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>black-box</td>
<td>naive</td>
</tr>
<tr>
<td>UCLID Suite:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aodv</td>
<td>2916</td>
<td>24</td>
<td>11</td>
</tr>
<tr>
<td>bakery</td>
<td>426</td>
<td>19</td>
<td>13</td>
</tr>
<tr>
<td>BRP</td>
<td>30</td>
<td>0.12</td>
<td>0.13</td>
</tr>
<tr>
<td>cache_ibm</td>
<td>326</td>
<td>2.3</td>
<td>2</td>
</tr>
<tr>
<td>cache_bounded</td>
<td>2238</td>
<td>63</td>
<td>31</td>
</tr>
<tr>
<td>DLX</td>
<td>30808</td>
<td>242</td>
<td>63</td>
</tr>
<tr>
<td>OOO</td>
<td>10728</td>
<td>176</td>
<td>57</td>
</tr>
</tbody>
</table>
Incremental refinement

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

<table>
<thead>
<tr>
<th>Benchmark family</th>
<th>#preds.</th>
<th>Time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>only $G_P(\varphi)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>step of 1</td>
</tr>
</tbody>
</table>

**UCLID Suite:**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#preds.</th>
<th>Step of 1</th>
<th>Step of 2</th>
<th>Step of 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>aodv</td>
<td>21</td>
<td>4.6</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>bakery</td>
<td>32</td>
<td>11</td>
<td>28</td>
<td>21</td>
</tr>
<tr>
<td>BRP</td>
<td>22</td>
<td>0.1</td>
<td>1.1</td>
<td>0.6</td>
</tr>
<tr>
<td>cache.ibm</td>
<td>16</td>
<td>1.3</td>
<td>3</td>
<td>2.2</td>
</tr>
<tr>
<td>cache.bounded</td>
<td>26</td>
<td>23</td>
<td>71</td>
<td>51</td>
</tr>
<tr>
<td>DLX</td>
<td>23</td>
<td>13</td>
<td>37</td>
<td>26</td>
</tr>
<tr>
<td>OOO</td>
<td>25</td>
<td>36</td>
<td>67</td>
<td>50</td>
</tr>
</tbody>
</table>

*Step of 2 means computing $G_P^2(\varphi), G_P^4(\varphi), \ldots, G_P^n(\varphi) \equiv G_P(\varphi)$*
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