Natasha Sharygina
www.verify.inf.usi.ch

Università della Svizzera Italiana (USI)

October 30, 2009
Outline

• FVS group at USI

• Group projects
  • Synergy of precise and fast abstraction
  • SMT-based decision procedures
  • Loop summarization

opensmt

LOOP FroG
Università della Svizzera Italiana (USI or University of Lugano) is located in the southernmost (and sunniest) part of Switzerland.

Members:

- Prof. Natasha Sharygina
- Postdoc: Roberto Bruttomesso
- PhD Students: Aliaksei Tsitovich, Simone Rollini
Synergy of precise and fast abstraction
Project motivation: existing approaches to abstraction in CEGAR loop are not perfect

Precise abstraction

- Minimal number of abstract transitions (no spurious transitions)

Fast abstraction

- Many ways to approximate the abstraction (Cartesian abstraction, predicate partitioning etc.)
- Usually very fast computation
- But...
  - Introduces spurious transitions (abstraction contains both spurious transitions and spurious paths)
  - Requires many refinement iterations to remove numerous spurious transitions.
Project motivation: existing approaches to abstraction in CEGAR loop are not perfect

Precise abstraction

- Minimal number of abstract transitions (no spurious transitions)
- Adding new predicates is enough to refine spurious path

Fast abstraction

- Many ways to approximate the abstraction (Cartesian abstraction, predicate partitioning etc.)
- Usually very fast computation
- But...
  - Introduces spurious transitions (abstraction contains both spurious transitions and spurious paths)
  - Requires many refinement iterations to remove numerous spurious transitions.
Project motivation: existing approaches to abstraction in CEGAR loop are not perfect

Precise abstraction

- Minimal number of abstract transitions (no spurious transitions)
- Adding new predicates is enough to refine spurious path
- But... Very slow computation (exponential in the number of predicates).
Project motivation: existing approaches to abstraction in CEGAR loop are not perfect

Precise abstraction

- Minimal number of abstract transitions (no spurious transitions)
- Adding new predicates is enough to refine spurious path
- But... Very slow computation (exponential in the number of predicates).

Fast abstraction

- Many ways to approximate the abstraction (Cartesian abstraction, predicate partitioning etc.)
Project motivation: existing approaches to abstraction in CEGAR loop are not perfect

**Precise abstraction**
- Minimal number of abstract transitions (no spurious transitions)
- Adding new predicates is enough to refine spurious path
- But... Very slow computation (exponential in the number of predicates).

**Fast abstraction**
- Many ways to approximate the abstraction (Cartesian abstraction, predicate partitioning etc.)
- Usually very fast computation
Project motivation: existing approaches to abstraction in CEGAR loop are not perfect

**Precise abstraction**
- Minimal number of abstract transitions (no spurious transitions)
- Adding new predicates is enough to refine spurious path
- But... Very slow computation (exponential in the number of predicates).

**Fast abstraction**
- Many ways to approximate the abstraction (Cartesian abstraction, predicate partitioning etc.)
- Usually very fast computation
- But...
  - Introduces spurious transitions (abstraction contains both spurious transitions and spurious paths)
  - Requires many refinement iterations to remove numerous spurious transitions.
Our solution: combine fast and precise predicate abstraction in CEGAR loop

Start with fast abstraction

Program → Abstraction → Model Checking → No violations

Counterexample $\pi$

Refinement ← Spurious ← Simulation → Real bug

Refine as precise as possible
Components of our algorithm

- **FastAbstraction**: given a set of predicates $\Pi$ and a concrete transition relation $T$ computes program *over-approximation* for $\hat{T}_\Pi$.

- **PreciseAbstraction**: given a set of predicates $\Pi$ and a concrete transition relation $T$ computes the *minimal abstraction* for $\hat{T}_\Pi$.

- **SpuriousTransition** ($\pi$): given a path $\pi$, maps every transition $t$ in $\pi$ to a set of predicates $P$, s.t. $P \subseteq \Pi$ and $t \not\in \hat{T}_P$.

- **SpuriousPath** ($\pi$): given a path $\pi$, maps every transition $t$ in $\pi$ to a set of predicates $P$, s.t. $\pi \not\in \hat{T}_{\sigma_{SP}}(t)$. Note that $\Pi \subseteq P$, i.e., SpuriousPath introduces new predicates.
The "synergy" algorithm

MixCegarLoop(\textit{TransitionSystem} \( M \), \textit{Property} \( F \))

\begin{align*}
\Pi &= \text{InitialPredicates}(F, T); \\
\alpha &= \text{FastAbstraction}(T, \Pi); \\
\text{while} \ not \ TIMEOUT \ do \\
\pi &= \text{ModelCheck}(\alpha, F); \\
\text{if} \ \pi = \emptyset \ \text{then} \ \text{return} \ \text{CORRECT}; \\
\text{else} \\
\sigma_{ST} &= \text{SpuriousTransition}(\pi); \\
\text{if} \ \sigma_{ST} \neq \emptyset \ \text{then} \\
\text{foreach} \ t \in \pi \ do \\
C &= \text{PreciseAbstraction}(T, \sigma_{ST}(t)); \\
\alpha &= \alpha \land C; \\
\text{else} \\
\sigma_{SP} &= \text{SpuriousPath}(\pi); \\
\text{if} \ \sigma_{SP} \neq \emptyset \ \text{then} \ \text{return} \ \text{INCORRECT}; \\
\text{else} \\
\text{foreach} \ t \in \pi \ do \\
\Pi &= \Pi \cup \sigma_{SP}(t); \\
C &= \text{PreciseAbstraction}(T, \sigma_{SP}(t)); \\
\alpha &= \alpha \land C;
\end{align*}
The “synergy” algorithm

MixCegarLoop(TransitionSystem \( M \), Property \( F \))
begin

\( \Pi = \text{InitialPredicates}(F,T) \);
\( \alpha = \text{FastAbstraction}(T,\Pi) \);

while not TIMEOUT do

\( \pi = \text{ModelCheck}(\alpha,F) \);
if \( \pi = \emptyset \) then return CORRECT;
else

\( \sigma_{ST} = \text{SpuriousTransition}(\pi) \);
if \( \sigma_{ST} \neq \emptyset \) then

foreach \( t \in \pi \) do

\( C = \text{PreciseAbstraction}(T,\sigma_{ST}(t)) \);
\( \alpha = \alpha \land C \);
else

\( \sigma_{SP} = \text{SpuriousPath}(\pi) \);
if \( \sigma_{SP} \neq \emptyset \) then return INCORRECT;
else

foreach \( t \in \pi \) do

\( \Pi = \Pi \cup \sigma_{SP}(t) \);
\( C = \text{PreciseAbstraction}(T,\sigma_{SP}(t)) \);
\( \alpha = \alpha \land C \);

end

Choose initial predicates \( \Pi \) and use them for fast abstraction.
The "synergy" algorithm

MixCegarLoop(TransitionSystem $M$, Property $F$)
begin
    $\Pi = \text{InitialPredicates}(F,T)$;
    $\alpha = \text{FastAbstraction}(T,\Pi)$;
    while not TIMEOUT do
        $\pi = \text{ModelCheck}(\alpha,F)$;
        if $\pi = \emptyset$ then return CORRECT;
        else
            $\sigma_{ST} = \text{SpuriousTransition}(\pi)$;
            if $\sigma_{ST} \neq \emptyset$ then foreach $t \in \pi$ do
                $C = \text{PreciseAbstraction}(T,\sigma_{ST}(t))$;
                $\alpha = \alpha \land C$;
            else
                $\sigma_{SP} = \text{SpuriousPath}(\pi)$;
                if $\sigma_{SP} \neq \emptyset$ then return INCORRECT;
                else foreach $t \in \pi$ do
                    $\Pi = \Pi \cup \sigma_{SP}(t)$;
                    $C = \text{PreciseAbstraction}(T,\sigma_{SP}(t))$;
                    $\alpha = \alpha \land C$;
            end
        end
    end
Perform Model Checking and obtain counterexample $\pi$ (if it exists)
The “synergy” algorithm

MixCegarLoop(TransitionSystem $M$, Property $F$)

\begin{align*}
\Pi &= \text{InitialPredicates}(F,T); \\
\alpha &= \text{FastAbstraction}(T,\Pi); \\
\text{while not TIMEOUT do} \\
\pi &= \text{ModelCheck}(\alpha,F); \\
\text{if } \pi = \emptyset \text{ then return CORRECT;} \\
\text{else} \\
\sigma_{\text{ST}} &= \text{SpuriousTransition}(\pi); \\
\text{if } \sigma_{\text{ST}} \neq \emptyset \text{ then} \\
\text{foreach } t \in \pi \text{ do} \\
C &= \text{PreciseAbstraction}(T,\sigma_{\text{ST}}(t)); \\
\alpha &= \alpha \land C; \\
\text{else} \\
\sigma_{\text{SP}} &= \text{SpuriousPath}(\pi); \\
\text{if } \sigma_{\text{SP}} \neq \emptyset \text{ then return INCORRECT;} \\
\text{else} \\
\text{foreach } t \in \pi \text{ do} \\
\Pi &= \Pi \cup \sigma_{\text{SP}}(t); \\
C &= \text{PreciseAbstraction}(T,\sigma_{\text{SP}}(t)); \\
\alpha &= \alpha \land C; \\
\end{align*}

Compute spurious transitions ($\sigma_{\text{ST}} : \forall t \in \pi \rightarrow P \subseteq \Pi \land t \not\models \hat{\top}_P$)
The “synergy” algorithm

MixCegarLoop(TransitionSystem \( M \), Property \( F \))

begin

\( \Pi = \text{InitialPredicates}(F,T); \)
\( \alpha = \text{FastAbstraction}(T,\Pi); \)

while not TIMEOUT do

\( \pi = \text{ModelCheck}(\alpha,F); \)
if \( \pi = \emptyset \) then return \text{CORRECT};
else

\( \sigma_{ST} = \text{SpuriousTransition}(\pi); \)
if \( \sigma_{ST} \neq \emptyset \) then

\[ \text{foreach } t \in \pi \text{ do} \]
\[ C = \text{PreciseAbstraction}(T,\sigma_{ST}(t)); \]
\[ \alpha = \alpha \land C; \]
else

\( \sigma_{SP} = \text{SpuriousPath}(\pi); \)
if \( \sigma_{SP} \neq \emptyset \) then return \text{INCORRECT};
else

\[ \text{foreach } t \in \pi \text{ do} \]
\[ \Pi = \Pi \cup \sigma_{SP}(t); \]
\[ C = \text{PreciseAbstraction}(T,\sigma_{SP}(t)); \]
\[ \alpha = \alpha \land C; \]
end

1. Perform Precise-Abstraction for predicates \( P \) related to spurious transitions \( \forall t \in \pi \).
2. Remove detected spurious transitions by refining original abstraction.

Note, all spurious transitions related to detected predicates are removed at once!
The “synergy” algorithm

\[ \text{MixCegarLoop}(\text{TransitionSystem } M, \text{ Property } F) \]
\begin{align*}
\text{begin} \\
\qquad \Pi = \text{InitialPredicates}(F,T); \\
\qquad \alpha = \text{FastAbstraction}(T,\Pi); \\
\qquad \text{while not TIMEOUT do} \\
\qquad \quad \pi = \text{ModelCheck}(\alpha,F); \\
\qquad \quad \text{if } \pi = \emptyset \text{ then return CORRECT; else} \\
\qquad \quad \sigma_{ST} = \text{SpuriousTransition}(\pi); \\
\qquad \quad \text{if } \sigma_{ST} \neq \emptyset \text{ then} \\
\qquad \quad \quad \text{foreach } t \in \pi \text{ do} \\
\qquad \quad \quad \quad C = \text{PreciseAbstraction}(T,\sigma_{ST}(t)); \\
\qquad \quad \quad \quad \alpha = \alpha \land C; \\
\qquad \quad \text{else} \\
\qquad \quad \quad \sigma_{SP} = \text{SpuriousPath}(\pi); \\
\qquad \quad \quad \quad \text{if } \sigma_{SP} \neq \emptyset \text{ then return INCORRECT; else} \\
\qquad \quad \quad \quad \text{foreach } t \in \pi \text{ do} \\
\qquad \quad \quad \quad \quad \Pi = \Pi \cup \sigma_{SP}(t); \\
\qquad \quad \quad \quad \quad C = \text{PreciseAbstraction}(T,\sigma_{SP}(t)); \\
\qquad \quad \quad \quad \quad \alpha = \alpha \land C; \\
\text{end}
\end{align*}
The “synergy” algorithm

MixCegarLoop(TransitionSystem $M$, Property $F$)
begin

$\Pi = \text{InitialPredicates}(F,T)$;
$\alpha = \text{FastAbstraction}(T,\Pi)$;
while not TIMEOUT do

$\pi = \text{ModelCheck}(\alpha,F)$;
if $\pi = \emptyset$ then return CORRECT;
else

$\sigma_{ST} = \text{SpuriousTransition}(\pi)$;
if $\sigma_{ST} \neq \emptyset$ then

foreach $t \in \pi$ do

$C = \text{PreciseAbstraction}(T,\sigma_{ST}(t))$;
$\alpha = \alpha \land C$;
else

$\sigma_{SP} = \text{SpuriousPath}(\pi)$;
if $\sigma_{SP} \neq \emptyset$ then return INCORRECT;
else

foreach $t \in \pi$ do

$\Pi = \Pi \cup \sigma_{SP}(t)$;
$C = \text{PreciseAbstraction}(T,\sigma_{SP}(t))$;
$\alpha = \alpha \land C$;
end
end

1. Add new predicates to $\Pi$ from Spurious-Path($\pi$).
2. Perform Precise-Abstraction for predicates $P$ related to transitions $\forall t \in \pi$.
3. Remove spurious path by refining the original abstraction.
Advantages of our algorithm

Summary:
Computes abstraction quickly but keeps it precise enough to avoid too many refinement iterations.
Advantages of our algorithm

Summary:
Computes abstraction quickly but keeps it precise enough to avoid too many refinement iterations

- Expensive precise abstraction is limited to a small number of predicates.
Advantages of our algorithm

Summary:
Computes abstraction quickly but keeps it precise enough to avoid too many refinement iterations

- Expensive precise abstraction is limited to a small number of predicates.
- Multiple spurious behaviors are removed at each refinement iteration (reduces CEGAR iterations)
Advantages of our algorithm

Summary:
Computes abstraction quickly but keeps it precise enough to avoid too many refinement iterations

- Expensive precise abstraction is limited to a small number of predicates.
- Multiple spurious behaviors are removed at each refinement iteration (reduces CEGAR iterations)
- Synergy can be localized to some parts of the program (for every location of the control-flow graph)
Experiments and ideas for future:

The “synergy” algorithm is implemented and evaluated in SATABSS software model checker — and it works. More details: http://verify.inf.usi.ch/projects/synergy.
Experiments and ideas for future:

The “synergy” algorithm is implemented and evaluated in SATABS software model checker — and it works.


Next:

1. Integrate synergy with interpolation-based approaches for predicate discovery.

2. Investigate trade-offs between precise and approximated approaches in the context of purely interpolation-based model checking.
SMT-based decision procedures
OpenSMT Overview

- SMT-Solvers are efficient tools to solve quantifier-free formulæ in some decidable logic

\[(a \lor (x + y \leq 0)) \land (\neg a \lor \neg b) \land (x + y \geq 10)\]
SMT-Solvers are efficient tools to solve quantifier-free formulæ in some decidable logic

\[(a \lor (x + y \leq 0)) \land (\neg a \lor \neg b) \land (x + y \geq 10)\]

**opensmt** is an **open-source** SMT-Solver with focus on

- **extensibility**: the SAT-to-theory interface is such that it is easy to plug-in new decision procedures
- **incrementality**: suitable for incremental verification
- **efficiency**: it is the fastest open-source solver for linear arithmetic, according to SMTCOMP’09
SMT-Solvers are efficient tools to solve quantifier-free formulæ in some decidable logic

\[(a \lor (x + y \leq 0)) \land (\neg a \lor \neg b) \land (x + y \geq 10)\]

opensmt is an open-source SMT-Solver with focus on

- **extensibility**: the SAT-to-theory interface is such that it is easy to plug-in new decision procedures
- **incrementality**: suitable for incremental verification
- **efficiency**: it is the fastest open-source solver for linear arithmetic, according to SMTCOMP’09

It combines the famous MINISAT2 SAT-Solver with state-of-the-art decision procedures for uninterpreted functions and predicates, linear arithmetic and bit-vector arithmetic
Motivations
(Other than doing research)

- to promote the use of SMT-Solvers in combination with other verification tools
Motivations
(other than doing research)

- to promote the use of SMT-Solvers in combination with other verification tools
- to provide a level of detail for decision procedures that goes beyond the scientific publication
Motivations
(Other than doing research)

- to promote the use of SMT-Solvers in combination with other verification tools
- to provide a level of detail for decision procedures that goes beyond the scientific publication
- to promote the development of SMT-Solvers by providing a simple infrastructure for the addition of new theories
Some distinguishing features of **OpenSMT**

- **Preprocessor for arithmetic SMT formulæ**
  - Implements a combination of the Davis-Putnam procedure and the Fourier-Motzkin elimination to simplify the formula at the preprocessing level.

---

FVS Group (USI)
Research Interests

October 30, 2009 14 / 21
Some distinguishing features of OpenSMT

- Preprocessor for arithmetic SMT formulæ
  - Implements a combination of the Davis-Putnam procedure and the Fourier-Motzkin elimination to simplify the formula at the preprocessing level
- An efficient and complete decision procedure for bit-vector extraction and concatenation
  - Reduces formulæ over bit-vector extraction and concatenation to the theory of equality, in order to avoid, when possible, a more expensive reduction to SAT
Other OpenSMT’s features

- C-API for integration with other verification frameworks
Other OpenSMT’s features

- C-API for integration with other verification frameworks
- Returns evidence of satisfiability (model) or unsatisfiability (proof - work in progress)
Other OpenSMT’s features

- C-API for integration with other verification frameworks
- Returns evidence of satisfiability (model) or unsatisfiability (proof - work in progress)
- Highly configurable via configuration file
Other OpenSMT’s features

- C-API for integration with other verification frameworks
- Returns evidence of satisfiability (model) or unsatisfiability (proof - work in progress)
- Highly configurable via configuration file

Project page: http://verify.inf.unisi.ch/opensmt
Code Repository: http://code.google.com/p/opensmt
Discussion Group: http://groups.google.com/group/opensmt
Program abstraction via loop summarization
Loops analysis — the Achilles’ heel of static analysis
Loops analysis — the Achilles’ heel of static analysis

- Loop unwinding is computationally too expensive (or even impossible) for many real programs.
Loops analysis — the Achilles’ heel of static analysis

- Loop unwinding is computationally too expensive (or even impossible) for many real programs.
- Loop over-approximation by computing its fixpoint is either too expensive to compute or too imprecise.
Loops analysis — the Achilles’ heel of static analysis

- Loop unwinding is computationally too expensive (or even impossible) for many real programs.
- Loop over-approximation by computing its fixpoint is either too expensive to compute or too imprecise.
- Loop over-approximation by discovering of sufficiently strong invariants is an art.
Loops analysis — the Achilles’ heel of static analysis

- Loop unwinding is computationally too expensive (or even impossible) for many real programs.
- Loop over-approximation by computing its fixpoint is either too expensive to compute or too imprecise.
- Loop over-approximation by discovering of sufficiently strong invariants is an art.

Multiple nested loops makes analysis even more difficult.
Our Solution

Avoid iterative computation of a loop abstract fixpoint. Instead build loop summaries. Make the summaries *precise*.
Our Solution

Avoid iterative computation of a loop abstract fixpoint. Instead build loop summaries. Make the summaries precise.

- Encode loop-free fragments into concrete summaries.
Avoid iterative computation of a loop abstract fixpoint. Instead build loop summaries. Make the summaries precise.

- Encode loop-free fragments into concrete summaries.
- Replace each loop by its abstract summary:
  - proceed bottom-up from the deep-most loop;
  - apply property-driven abstract domains to obtain localized invariant candidates for each loop;
  - use the concrete symbolic transformer of a loop body to check if it is a loop invariant;
  - construct a loop summary as a combination of loop variants and discovered invariants.
Avoid iterative computation of a loop abstract fixpoint. Instead build loop summaries. Make the summaries precise.

- Encode loop-free fragments into concrete summaries.
- Replace each loop by its abstract summary:
  - proceed bottom-up from the deep-most loop;
  - apply property-driven abstract domains to obtain localized invariant candidates for each loop;
  - use the concrete symbolic transformer of a loop body to check if it is a loop invariant;
  - construct a loop summary as a combination of loop variants and discovered invariants.
- Perform an assertion check on the obtained abstract model. Since there are no loops anymore, expensive iterative computation is avoided.
Implementation

**LOOPFROG**- static analysis tool for C programs

- Works on models from ANSI-C programs that are created using Goto-CC front-end\(^1\);
- Uses SAT-based symbolic engine of CBMC for invariant candidates check and final assertion check;
- Performs sound and scalable loop summarization.

\(^1\)http://www.cprover.org/goto-cc
Current results and future work

- Loopfrog provides a library of abstract domains tailored to verification of safety of string operations in C.
- It was applied not just to crafted benchmarks but to real large-scale open-source software like GNUPG, INN, and WU-FTPD.

Project page: http://verify.inf.unisi.ch/loopfrog
Current results and future work

- Loopfrog provides a library of abstract domains tailored to verification of safety of string operations in C.
- It was applied not just to crafted benchmarks but to real large-scale open-source software like GNUPG, INN, and WU-FTPD.

Project page: http://verify.inf.unisi.ch/loopfrog

Next:
- Combine loop summarization with various invariant discovery methods;
- Employ SMT-Solver based decision back-end for more expressive invariant candidates and faster checks.
Thank you!