Formal Verification and Security Group

Research Interests

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 opensmt
    - Loop summarization LoopFroG

Formal Verification and Security Group at USI

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

Formal Verification and Security Group at USI

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)

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

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

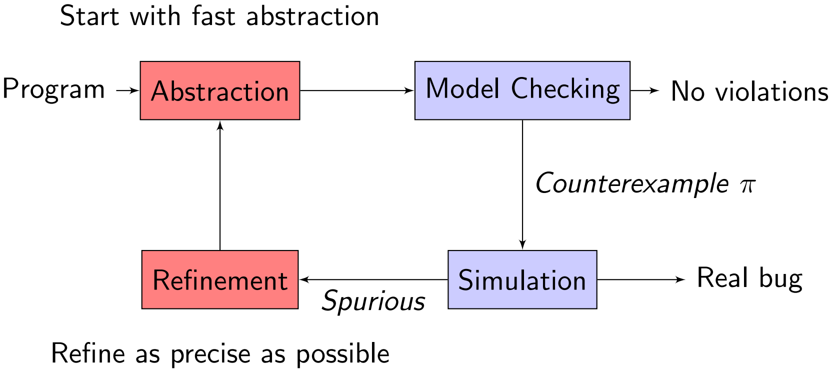
Precise abstraction

- 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

Refine as precise as possible

Components of our algorithm

- `FastAbstraction`: given a set of predicates Π and a concrete transition relation T computes program *over-approximation* for $\hat{T}_{\Pi}$.
- `PreciseAbstraction`: given a set of predicates Π and a concrete transition relation T computes the *minimal abstraction* for $\hat{T}_{\Pi}$.
- `SpuriousTransition` (π): given a path π, maps every transition t in π to a set of predicates P, s.t. $P \subseteq \Pi$ and $t \not\models \hat{T}_P$.
- `SpuriousPath` (π): given a path π, maps every transition t in π to a set of predicates P, s.t. $\pi \not\models \hat{T}_{\sigma_{SP}(t)}$. Note that $\Pi \subseteq P$, i.e., `SpuriousPath` introduces new predicates.

The “synergy” algorithm

Let’s proceed stepwise

```
MixCegarLoop(TransitionSystem M, Property F)
begin
    $\Pi$ = InitialPredicates($F$,$T$);
    $\alpha$ = FastAbstraction($T$,$\Pi$);
    while not TIMEOUT do
        $\pi$ = ModelCheck($\alpha$,$F$);
        if $\pi = \emptyset$ then return CORRECT;
        else
            $\sigma_{ST}$ = SpuriousTransition($\pi$);
            if $\sigma_{ST} \neq \emptyset$ then
                foreach $t \in \pi$ do
                    $C$ = PreciseAbstraction($T$,$\sigma_{ST}(t)$);
                    $\alpha = \alpha \wedge C$;
            else
                $\sigma_{SP}$ = SpuriousPath($\pi$);
                if $\sigma_{SP} \neq \emptyset$ then return INCORRECT;
                else
                    foreach $t \in \pi$ do
                        $\Pi = \Pi \cup \sigma_{SP}(t)$;
                        $C$ = PreciseAbstraction($T$,$\sigma_{SP}(t)$);
                        $\alpha = \alpha \wedge C$;
end
```

The “synergy” algorithm

```
MixCegarLoop(TransitionSystem M, Property F)
begin
    $\Pi$ = InitialPredicates($F$,$T$);
    $\alpha$ = FastAbstraction($T$,$\Pi$);
    while not TIMEOUT do
        $\pi$ = ModelCheck($\alpha$,$F$);
        if $\pi = \emptyset$ then return CORRECT;
        else
            $\sigma_{ST}$ = SpuriousTransition($\pi$);
            if $\sigma_{ST} \neq \emptyset$ then
                foreach $t \in \pi$ do
                    $C$ = PreciseAbstraction($T$,$\sigma_{ST}(t)$);
                    $\alpha = \alpha \wedge C$;
            else
                $\sigma_{SP}$ = SpuriousPath($\pi$);
                if $\sigma_{SP} \neq \emptyset$ then return INCORRECT;
                else
                    foreach $t \in \pi$ do
                        $\Pi = \Pi \cup \sigma_{SP}(t)$;
                        $C$ = PreciseAbstraction($T$,$\sigma_{SP}(t)$);
                        $\alpha = \alpha \wedge C$;
end
```

Choose initial predicates Π and use them for fast abstraction

The “synergy” algorithm

```
MixCegarLoop(TransitionSystem M, Property F)
begin
    $\Pi$ = InitialPredicates($F$,$T$);
    $\alpha$ = FastAbstraction($T$,$\Pi$);
    while not TIMEOUT do
        $\pi$ = ModelCheck($\alpha$,$F$);
        if $\pi = \emptyset$ then return CORRECT;
        else
            $\sigma_{ST}$ = SpuriousTransition($\pi$);
            if $\sigma_{ST} \neq \emptyset$ then
                foreach $t \in \pi$ do
                    $C$ = PreciseAbstraction($T$,$\sigma_{ST}(t)$);
                    $\alpha = \alpha \wedge C$;
            else
                $\sigma_{SP}$ = SpuriousPath($\pi$);
                if $\sigma_{SP} \neq \emptyset$ then return INCORRECT;
                else
                    foreach $t \in \pi$ do
                        $\Pi = \Pi \cup \sigma_{SP}(t)$;
                        $C$ = PreciseAbstraction($T$,$\sigma_{SP}(t)$);
                        $\alpha = \alpha \wedge C$;
end
```

Perform Model Checking and obtain counterexample π (if it exists)

The "synergy" algorithm

```
MixCegarLoop(TransitionSystem M, Property F)
begin
    $\Pi$ = InitialPredicates($F$,$T$);
    $\alpha$ = FastAbstraction($T$,$\Pi$);
    while not TIMEOUT do
        $\pi$ = ModelCheck($\alpha$,$F$);
        if $\pi = \emptyset$ then return CORRECT;
        else
            $\sigma_{ST}$ = SpuriousTransition($\pi$);
            if $\sigma_{ST} \neq \emptyset$ then
                foreach $t \in \pi$ do
                    $C$ = PreciseAbstraction($T$,$\sigma_{ST}(t)$);
                    $\alpha = \alpha \wedge C$;
            else
                $\sigma_{SP}$ = SpuriousPath($\pi$);
                if $\sigma_{SP} \neq \emptyset$ then return INCORRECT;
                else
                    foreach $t \in \pi$ do
                        $\Pi = \Pi \cup \sigma_{SP}(t)$;
                        $C$ = PreciseAbstraction($T$,$\sigma_{SP}(t)$);
                        $\alpha = \alpha \wedge C$;
end
```

Compute spurious transitions ($\sigma_{ST} : \forall t \in \pi \rightarrow P \subseteq \Pi \wedge t \not\models \hat{T}_P$)

The “synergy” algorithm

```
MixCegarLoop(TransitionSystem M, Property F)
begin
    $\Pi$ = InitialPredicates($F$,$T$);
    $\alpha$ = FastAbstraction($T$,$\Pi$);
    while not TIMEOUT do
        $\pi$ = ModelCheck($\alpha$,$F$);
        if $\pi = \emptyset$ then return CORRECT;
        else
            $\sigma_{ST}$ = SpuriousTransition($\pi$);
            if $\sigma_{ST} \neq \emptyset$ then
                foreach $t \in \pi$ do
                    $C$ = PreciseAbstraction($T$,$\sigma_{ST}(t)$);
                    $\alpha = \alpha \wedge C$;
            else
                $\sigma_{SP}$ = SpuriousPath($\pi$);
                if $\sigma_{SP} \neq \emptyset$ then return INCORRECT;
                else
                    foreach $t \in \pi$ do
                        $\Pi = \Pi \cup \sigma_{SP}(t)$;
                        $C$ = PreciseAbstraction($T$,$\sigma_{SP}(t)$);
                        $\alpha = \alpha \wedge C$;
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

```
MixCegarLoop(TransitionSystem M, Property F)
begin
    $\Pi$ = InitialPredicates($F$,$T$);
    $\alpha$ = FastAbstraction($T$,$\Pi$);
    while not TIMEOUT do
        $\pi$ = ModelCheck($\alpha$,$F$);
        if $\pi = \emptyset$ then return CORRECT;
        else
            $\sigma_{ST}$ = SpuriousTransition($\pi$);
            if $\sigma_{ST} \neq \emptyset$ then
                foreach $t \in \pi$ do
                    $C$ = PreciseAbstraction($T$,$\sigma_{ST}(t)$);
                    $\alpha = \alpha \wedge C$;
            else
                $\sigma_{SP}$ = SpuriousPath($\pi$);
                if $\sigma_{SP} \neq \emptyset$ then return INCORRECT;
                else
                    foreach $t \in \pi$ do
                        $\Pi = \Pi \cup \sigma_{SP}(t)$;
                        $C$ = PreciseAbstraction($T$,$\sigma_{SP}(t)$);
                        $\alpha = \alpha \wedge C$;
end
```

Otherwise check if π has any spurious path $(\sigma_{SP} : t \in \pi \rightarrow \Pi \subseteq P \wedge \pi \not\models \hat{T}_{\sigma_{SP}(t)})$

The "synergy" algorithm

```
MixCegarLoop(TransitionSystem M, Property F)
begin
    $\Pi$ = InitialPredicates($F$,$T$);
    $\alpha$ = FastAbstraction($T$,$\Pi$);
    while not TIMEOUT do
        $\pi$ = ModelCheck($\alpha$,$F$);
        if $\pi = \emptyset$ then return CORRECT;
        else
            $\sigma_{ST}$ = SpuriousTransition($\pi$);
            if $\sigma_{ST} \neq \emptyset$ then
                foreach $t \in \pi$ do
                    $C$ = PreciseAbstraction($T$,$\sigma_{ST}(t)$);
                    $\alpha = \alpha \wedge C$;
            else
                $\sigma_{SP}$ = SpuriousPath($\pi$);
                if $\sigma_{SP} \neq \emptyset$ then return INCORRECT;
                else
                    foreach $t \in \pi$ do
                        $\Pi = \Pi \cup \sigma_{SP}(t)$;
                        $C$ = PreciseAbstraction($T$,$\sigma_{SP}(t)$);
                        $\alpha = \alpha \wedge C$;
end
```

1. Add new predicates to Π from `Spurious-Path(π)`.
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 SATABS 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.

More details: `http://verify.inf.usi.ch/projects/synergy`.

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

$$(a \vee (x + y \leq 0)) \wedge (\neg a \vee \neg b) \wedge (x + y \geq 10)$$

OpenSMT Overview

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

$$(a \vee (x + y \leq 0)) \wedge (\neg a \vee \neg b) \wedge (x + y \geq 10)$$

- 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

OpenSMT Overview

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

$$(a \vee (x + y \leq 0)) \wedge (\neg a \vee \neg b) \wedge (x + y \geq 10)$$

- 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

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

Project Motivation

Loops analysis — the Achilles' heel of static analysis

Project Motivation

Loops analysis — the Achilles' heel of static analysis

- Loop unwinding is computationally too expensive (or even impossible) for many real programs.

Project Motivation

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.

Project Motivation

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.

Project Motivation

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.

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.
- 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.

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.
- 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!