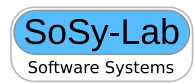
A Unifying Approach for Control-Flow-Based Loop Abstraction

Dirk Beyer, Marian Lingsch, and **Martin Spiessl**

LMU Munich, Germany

2022-09-13



Motivational Example: Naive Loop Abstraction

```
1  void main() {
2    int i = 0;
3    while (i<N) {
4      i=i+1;
5    }
6    assert (i==N);
7  }
```

Motivational Example: Naive Loop Abstraction

```
1  void main() {
2    int i = 0;
3    while (i<N) {
4      i=i+1;
5    }
6    assert (i==N);
7  }
```

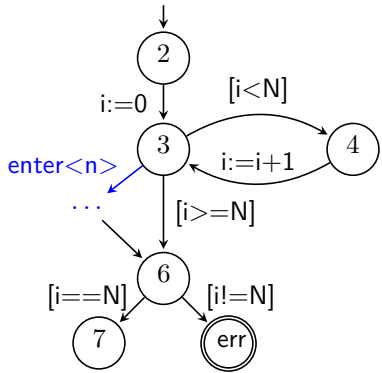
```
1  void main() {
2    int i = 0;
3    if (i<N) {
4      i = nondet();
5      assume(i<N);
6      i=i+1;
7      assume(!(i<N));
8    }
9    assert (i==N);
10 }
```

- Naive Loop Abstraction [5]:
  havoc all input variables of the loop and perform one loop iteration
- Only sound if the loop body does not contain assertions
- Overapproximation, but sometimes enough (like in this example)

Motivational Example: Naive Loop Abstraction

```
1  void main() {
2    int i = 0;
3    while (i<N) {
4      i=i+1;
5    }
6    assert (i==N);
7  }
```

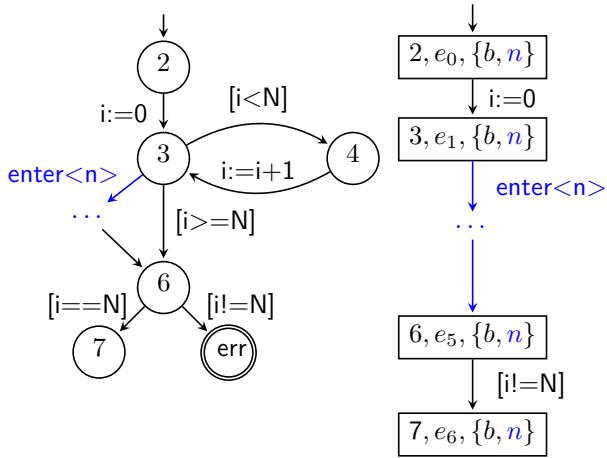
```
1  void main() {
2    int i = 0;
3    if (i<N) {
4      i = nondet();
5      assume(i<N);
6      i=i+1;
7      assume(!(i<N));
8    }
9    assert (i==N);
10 }
```



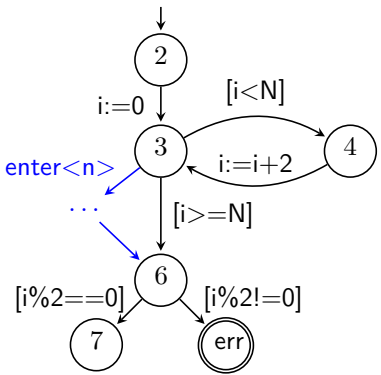
- Naive Loop Abstraction [5]:
  havoc all input variables of the loop and perform one loop iteration
- Only sound if the loop body does not contain assertions
- Overapproximation, but sometimes enough (like in this example)

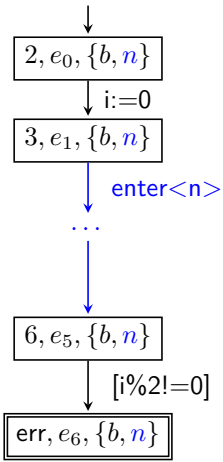
Control-Flow-Based Loop Abstraction: Example 1

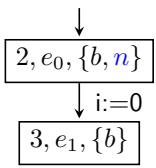
- Once reaching location 3, we follow the naive loop abstraction strategy
- The proof succeeds
- Otherwise (see next slide):
  - Backtrack
  - Update precision
  - Here this means: analyze original program

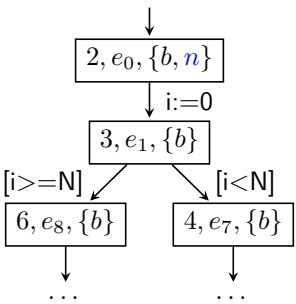


Control-Flow-Based Loop Abstraction: Example 2





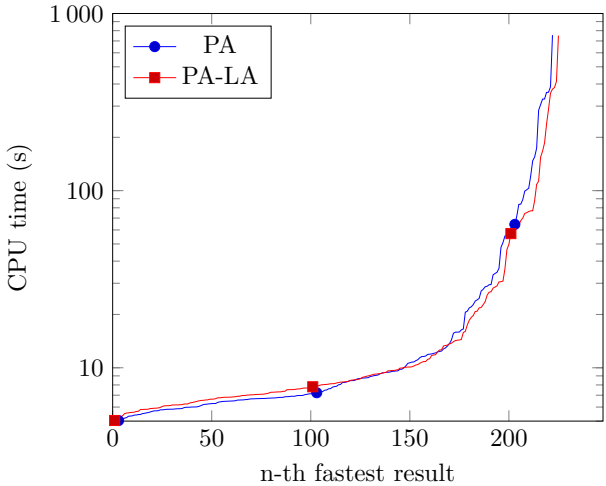




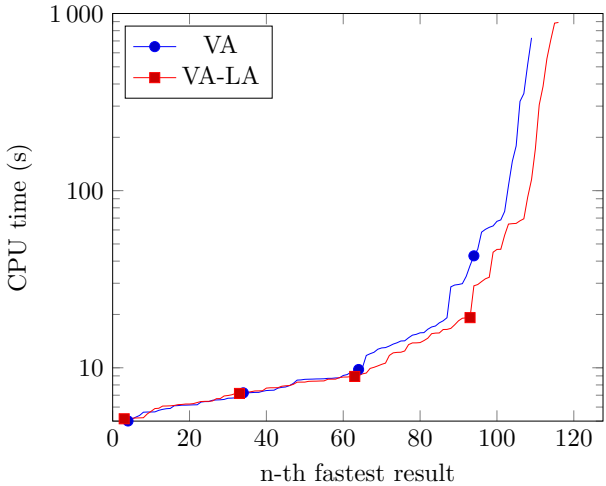
Evaluation

- Benchmark tasks: ReachSafety-Loops from SV-Benchmarks (765 tasks)
- Resource limits: CPU time 900 s, 15 GB RAM, 2 processing units
- Considered analyses in CPAchecker:
  - Predicate Abstraction (PA)
  - Value Analysis (VA)
  - Bounded Model Checking (BMC)
- Used loop abstractions: havoc, naive abstraction[3], constant extrapolation, output abstraction[2]
- Question: can we improve these analyses with our loop abstraction approach?

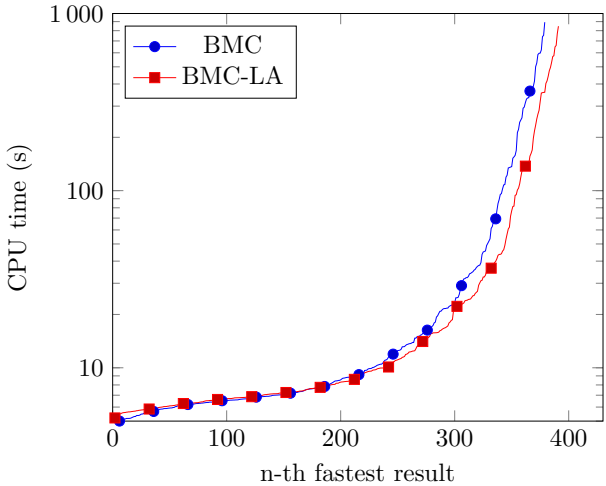
Results for Predicate Abstraction



Results for Value Analysis



Results for Bounded Model Checking



Accessibility of Loop Abstractions via Patches

- We provide loop abstractions as patches
- Can be used independently by other tools
- Experiments show this can help other verifiers like CBMC (cf. full paper)

```
--- havoc.c
+++ havoc.c
-14,13 +14,16
   return;
 }

 int main(void) {
   unsigned int x = 1000000;
- while (x > 0) {
- x -= 4;
+ // START HAVOCSTRATEGY
+ if (x > 0) {
+ x = __VERIFIER_nondet_uint();
   }
+ if (x > 0) abort();
+ // END HAVOCSTRATEGY
   __VERIFIER_assert(!(x % 4));
```

Contribution

- Novel CEGAR approach for applying loop abstractions
- Independent of the underlying abstract domain
- Implemented in the CPAchecker framework
- Easily extensible with new abstraction strategies
- Loop abstractions are made available via patches
- Evaluated on benchmarks from the SV-Benchmarks set

Preprint PDF:

References I

- Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language version 1.15 (2020)
- Darke, P., Chimdyalwar, B., Venkatesh, R., Shrotri, U., Metta, R.: Over-approximating loops to prove properties using bounded model checking. In: Proc. DATE. pp. 1407–1412. IEEE (2015). https://doi.org/10.7873/DATE.2015.0245
- Darke, P., Khanzode, M., Nair, A., Shrotri, U., Venkatesh, R.: Precise analysis of large industry code. In: Proc. APSEC. pp. 306–309. IEEE (2012). https://doi.org/10.1109/APSEC.2012.97
- Darke, P., Chimdyalwar, B., Venkatesh, R., Shrotri, U., Metta, R.: Over-approximating loops to prove properties using bounded model checking. In: Nebel, W., Atienza, D. (eds.) Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015. pp. 1407–1412. ACM (2015)
- Darke, P., Khanzode, M., Nair, A., Shrotri, U., Venkatesh, R.: Precise analysis of large industry code. In: Leung, K.R.P.H., Muenchaisri, P. (eds.) 19th Asia-Pacific Software Engineering Conference, APSEC 2012, Hong Kong, China, December 4-7, 2012. pp. 306–309. IEEE (2012). https://doi.org/10.1109/APSEC.2012.97

References II

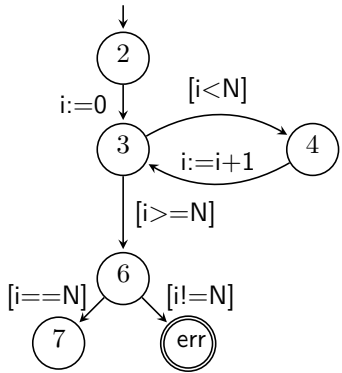
Kumar, S., Sanyal, A., Venkatesh, R., Shah, P.: Property checking array programs using loop shrinking. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10805, pp. 213–231. Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_12

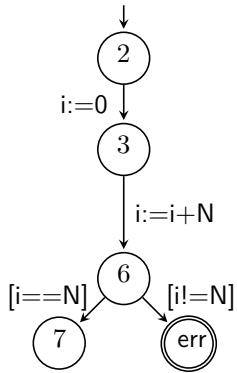
Loop Acceleration vs. Loop Abstraction

- **Loop Acceleration**:
  describes techniques that calculate the precise effect of a loop
- **Loop Abstraction**:
  describes techniques that overapproximate the semantics of a loop
- We can treat Loop Acceleration as a special case of Loop Abstraction
  $\Rightarrow$In this talk we will refer to both as Loop Abstractions

Introductory Example: Loop Acceleration

```
1  void main() {
2    int i = 0;
3    while (i<N) {
4      i=i+1;
5    }
6    assert (i==N);
7  }
```

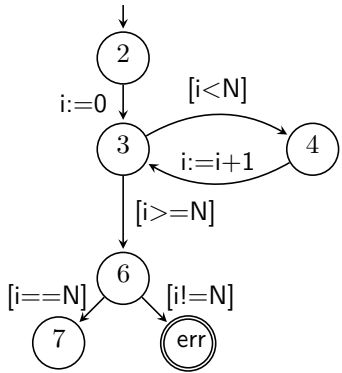


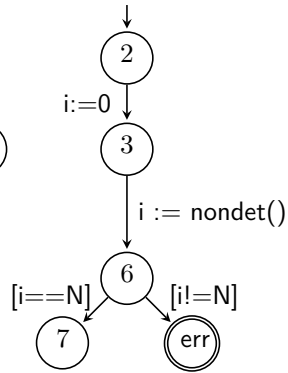


- Unrolling the loop for verification is often prohibitively expensive for large N
- Simple cases like the one shown here can be *accelerated*
- Downside: Traces do not correspond to the original program any more

Introductory Example: Loop Abstraction

```
1  void main() {
2    int i = 0;
3    while (i<N) {
4      i=i+1;
5    }
6    assert (i==N);
7  }
```





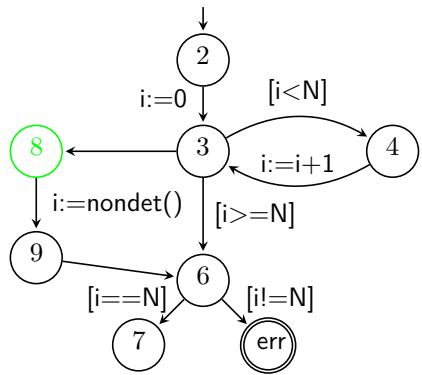
- Instead of a precise acceleration, we can also apply an overapproximating *abstraction*
- Here we just havoc all variables that are modified in the loop, but more elaborate abstraction strategies exist

Motivation

- many *loop abstraction strategies* exist:
  - extrapolation
  - naive loop abstraction
  - k-induction
  - ...
- Usually these are applied as source code transformation
- No single tool exists that implements all of them and enables a comparison
- $\Rightarrow$ We want to be able to:
  - Compare them all inside a single framework
  - Decide during the state-space exploration which strategies work for the verification problem at hand (using CEGAR)
  - Be able to map our verification results back to the original program

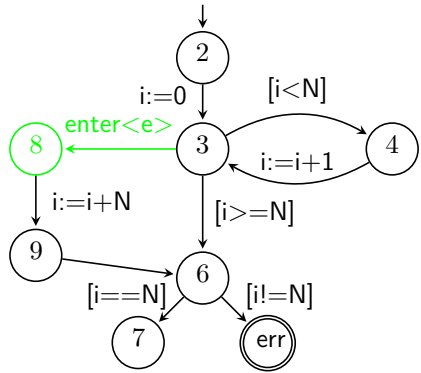
Proposed Solution

- Use the CFA as interface
- Add our loop abstractions next to the original loop
- Mark the entry nodes of each added alternative with an identifier for the applied strategy: $\sigma : L \to S$
- In the example:
  $S = \{b, h\}$
  $\sigma(8) = h$
  $\sigma(l) = b$ for $l \in \{2, 3, 4, 6, 7, err, 9\}$
- Select allowed strategies during state-space exploration using σ



Extrapolation Strategy

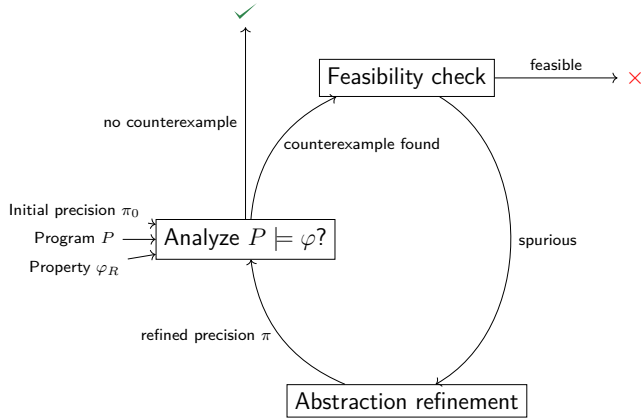
```
1  void main() {
2    int i = 0;
3    while (i<N) {
4      i=i+1;
5    }
6    assert (i==N);
7  }
```



- This is a precise abstraction, i.e., an acceleration
- We are not limited to precise abstractions, we can also use overapproximations like naive loop abstraction (next slide)

CEGAR: Feasibility of Counterexamples

- In general, CEGAR works as shown on the right
- For our approach, we need to rethink what it means if a counterexample is feasible: Even if the path formula is satisfiable, the counterexample is only feasible if there are no over-approximating strategies used along the path!



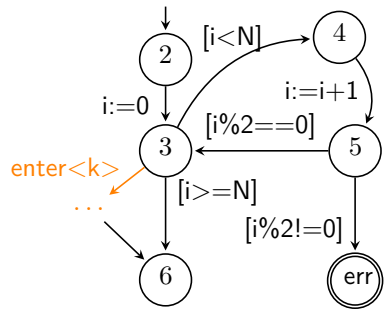
CEGAR: Refinement Chaining

- Question: How does this refinement interfere with the regular CEGAR refinement of the analysis we use?
- Answer: This is completely transparent and does not affect the inner CEGAR refinement
- The refinement operator modifies the reached set and waitlist:
  $\texttt{refine} : (\text{reached}, \texttt{waitlist}) \mapsto (\text{reached}', \texttt{waitlist}')$
  $\text{reached}, \texttt{waitlist} \subseteq L \times E \times \Pi$
- $\Rightarrow$ We can chain our strategy precision refinement $\texttt{refine}_{\mathbb{S}}$ with the refinement $\texttt{refine}_{\mathbb{W}}$ of the wrapped analysis:
  $\texttt{refine} = \texttt{refine}_{\mathbb{S}} \circ \texttt{refine}_{\mathbb{W}}$

Abstraction Using (k-)Induction

```
1  void main() {
2    int i = 0;
3    while (i<N) {
4      i=i+2;
5      assert(i%2==0);
6    }
7  }
```

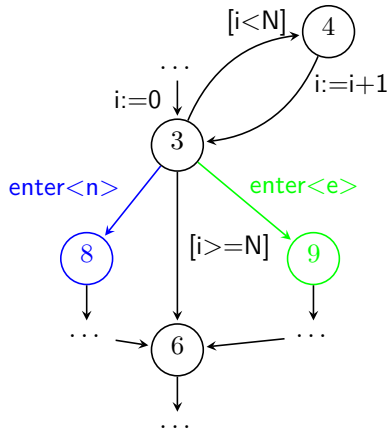
```
1  void main() {
2    int i = 0;
3    if (i<N) { // base case
4    i=i+2;
5    assert(i%2==0);
6    }
7    havoc(i);
8    i=i+2; // step case
9    assume(i%2==0);
10   if (i<N) {
11   i+=2;
12   assert(i%2==0);
13   }}
```



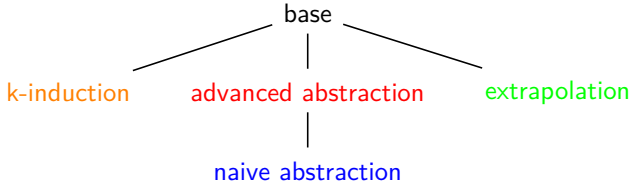
- Technique is successfully used in existing tools [4]
- Makes use of the fact that the property itself is (k-)inductive
- We can improve upon this by adding support for auxiliary invariants

Choice of Allowed Successors

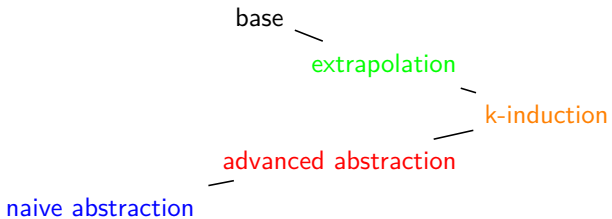
- Imagine we are at node 3 in the CFA on the right
- We have to decide which successors to generate
- Available strategies form the set A, e.g. here in node 3: $A = \{b, n, e\}$
- Allowed successors will be determined by the function `select`, which needs to satisfy: $\texttt{select}(A, \pi_{\mathbb{S}}) \subseteq A \setminus \pi_{\mathbb{S}}$
- Function `select` can be induced by any strict total or partial order $\sqsubset$ over S: $\texttt{select}(A, \pi_{\mathbb{S}}) = \{s \in A \setminus \pi_{\mathbb{S}} \mid \nexists s' \in A : s' \sqsubset s\}$



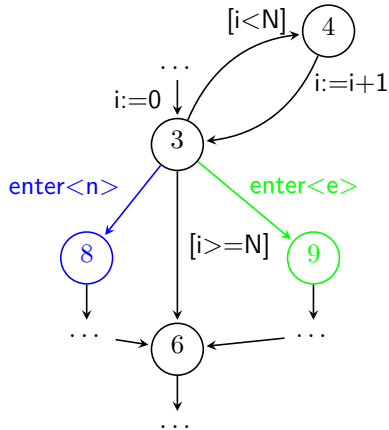
Examples for Orders over Abstraction Strategies



- `select`$(\{b, n, e\}, \{\}) = \{n, e\}$



- `select`$(\{b, n, e\}, \{\}) = \{n\}$



State-Space Exploration

- In the following examples, we will show abstract states as triples $a = (l, e, \pi_{\mathbb{S}})$
  - l is the current location in the CFA
  - e is the abstract state (depending on analysis)
  - $\pi_{\mathbb{S}}$ is the strategy precision for selection
- Example: $a = (3, e_2, \{n, e\})$
- In our transfer relation we will need to decide which strategies to apply based the function `select`

Some of the Planned Additions

- Use a location-based strategy precision instead of a global one
- Add a k-induction strategy with the possibility to use externally provided invariants (use cases: interactive verification, witness validation)
- Extend the witness format to include information about the used acceleration strategies
- Add acceleration of loops with array accesses, e.g. via k-shrinkability [6]
- Recursion: as starting point, a strategy to detect end-recursive procedure calls and rewrite them into iterative form should be simple to implement
- Witness Generation: map our reachability graph over the strategy-augmented CFA back to a witness automaton over the original program's CFA
- Add support for (ACSL) function contracts

Outlook: Function Contracts

```
1  /*@ requires 0<=n<65536 && *res==0;
2   *@ assigns *res;
3   *@ ensures *res == n*(n+1)/2; */
4  void sum(int n, int *res) {
5    while (n>0) {*res+=n;n--;}
6  }
7  void main() {
8    int i = 0;
9
10
11   sum(1000,&i);
12
13   assert(i==500*1001);
14 }
```

```
1  /*@ requires 0<=n<65536 && *res==0;
2   *@ assigns *res;
3   *@ ensures *res == n*(n+1)/2; */
4  void sum(int n, int *res) {
5    while (n>0) {*res+=n;n--;}
6  }
7  void main() {
8    int i = 0;
9    assert(0<=1000 && 1000<65536);
10   assert(i==0);
11   havoc(i);
12   assume(i==1000*(1000+1)/2);
13   assert(i==500500);
14 }
```

- We can replace function calls
  in case a function contract (e.g. written in ACSL [1]) is provided
- The function contract can be verified separately