A Unifying Approach for Control-Flow-Based Loop Abstraction

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

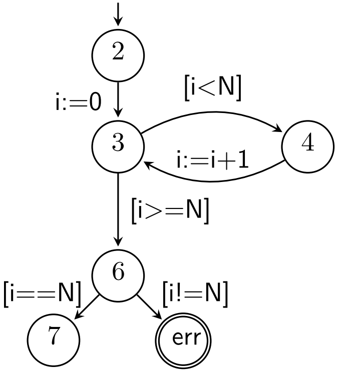
LMU Munich, Germany

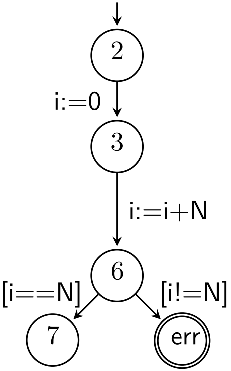
Loop Acceleration vs. Loop Abstraction

- **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
  ⇒In this talk we will refer to both as Loop Abstractions

Introductory Example: Loop Acceleration

```
void main() {
  int i = 0;
  while (i<N) {
    i=i+1;
  }
  assert (i==N);
}
```

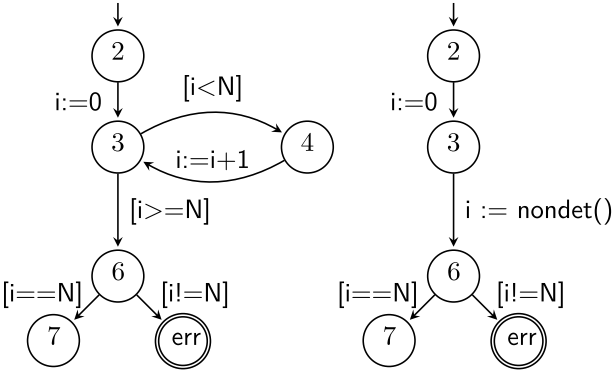




- Unrolling the loop for verification is often prohibitively expensive for large N
- Simple cases like the one shown here can be *accelerated*
- Downside: Traces do not correspond to the original program any more

Introductory Example: Loop Abstraction

```
void main() {
  int i = 0;
  while (i<N) {
    i=i+1;
  }
  assert (i==N);
}
```



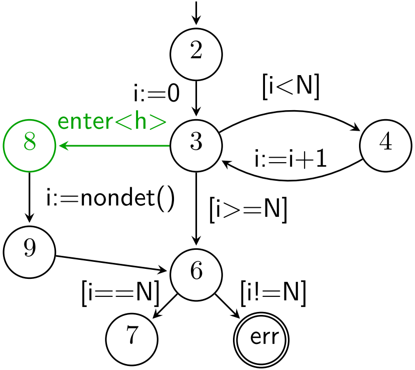
- Instead of a precise acceleration, we can also apply an overapproximating *abstraction*
- Here we just havoc all variables that are modified in the loop, but more elaborate abstraction strategies exist

Motivation

- many *loop abstraction strategies* exist:
  - constant extrapolation
  - havoc abstraction
  - ...
- Usually these are applied as source code transformation
- No single tool exists that implements all of them and enables a comparison
- ⇒ We want to be able to:
  - **Compare** them all inside a single framework
  - **Select** during the state-space exploration which strategies work for the verification problem at hand (using CEGAR)
  - **Map** our verification results back to the original program
  - **Reuse** loop abstractions by making them available via patches

Proposed Solution

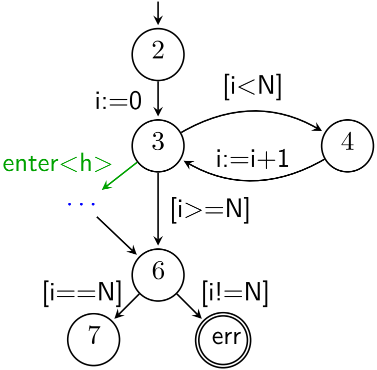
- Use the CFA as interface
- Add our loop abstractions next to the original loop
- Mark the entry nodes of each added alternative with an identifier for the applied strategy: $\sigma : L \to S$
- In the example:
  $S = \{b, h\}$
  $\sigma(8) = h$
  $\sigma(l) = b$ for $l \in \{2, 3, 4, 6, 7, err, 9\}$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



Havoc Abstraction

```
void main() {
  int i = 0;
  while (i<N) {
    i=i+1;
  }
  assert (i==N);
}
```

```
void main() {
  int i = 0;
  if (i<N) {
    i = nondet();
    assume(!(i<N));
  }
  assert (i==N);
}
```

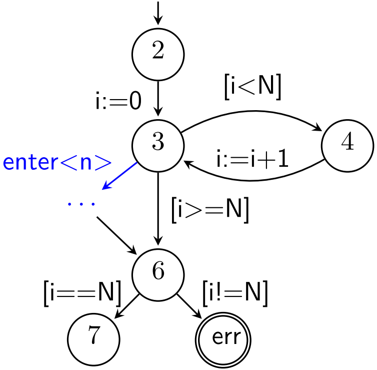


- Havoc Abstraction: if loop is entered, havoc all input variables of the loop and perform one loop iteration, then assume the loop is left
- Only sound if the loop body does not contain assertions
- Overapproximation, but sometimes enough (not in this example)

Naive Loop Abstraction

```
void main() {
  int i = 0;
  while (i<N) {
    i=i+1;
  }
  assert (i==N);
}
```

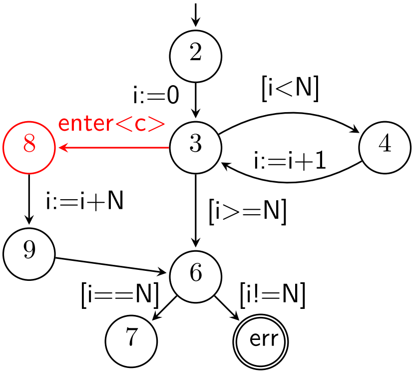
```
void main() {
  int i = 0;
  if (i<N) {
    i = nondet();
    assume(i<N);
    i=i+1;
    assume(!(i<N));
  }
  assert (i==N);
}
```



- Naive Loop Abstraction [4]:
  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)

Constant Extrapolation Strategy

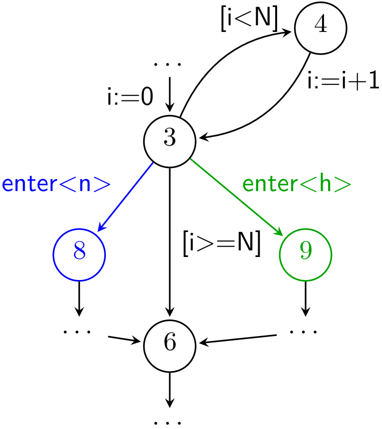
```
void main() {
  int i = 0;
  while (i<N) {
    i=i+1;
  }
  assert (i==N);
}
```



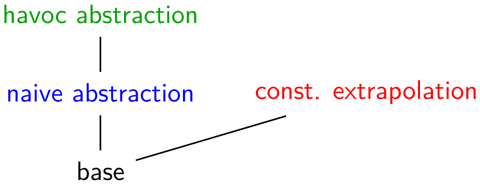
- Constant Extrapolation: For loops with a finite bound that only increments variables by a constant, the end result can be easily computed
- This is a precise abstraction, i.e., an acceleration

Choice of Allowed Successors

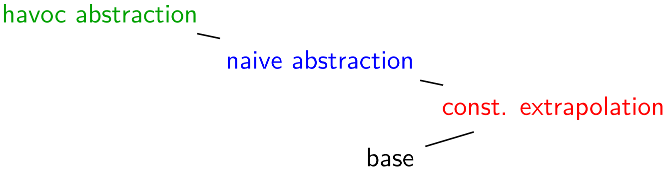
- Imagine we are at node 3 in the CFA on the right
- We have to decide which successors to generate
- Available strategies form the set A, e.g. here in node 3: $A = \{b, n, h\}$
- Allowed strategies are tracked in the set $\pi_{\mathbb{S}}$
- Allowed successors will be determined by the function `select`,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



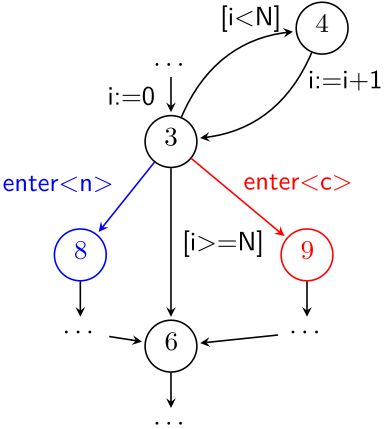
Examples for Orders over Abstraction Strategies



- $\texttt{select}(\{b, n, c\}, \{b, n, c, h\}) = \{n, c\}$



- $\texttt{select}(\{b, n, c\}, \{b, n, c, h\}) = \{n\}$

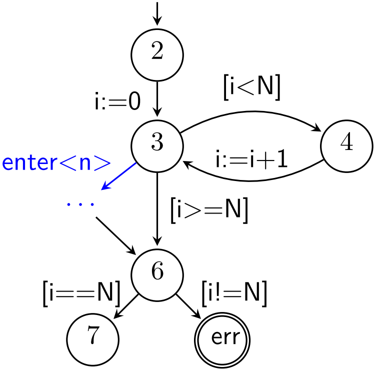


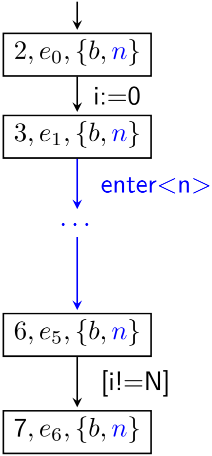
State-Space Exploration

- 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: $a = (3, e_2, \{b, n, c\})$
- In our transfer relation we will need to decide which strategies to apply based the function `select`

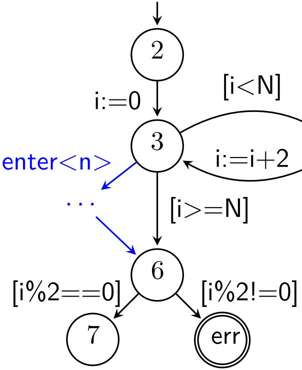
Loop Abstraction with CEGAR: Example 1

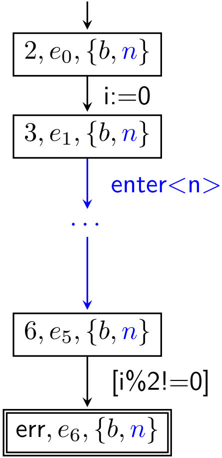
- Once reaching location 3, we follow the naive loop abstraction strategy
- The proof succeeds
- Otherwise (see next slide):
    - Backtrack
    - Update precision
    - Here this means: analyze original program

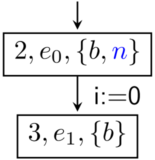


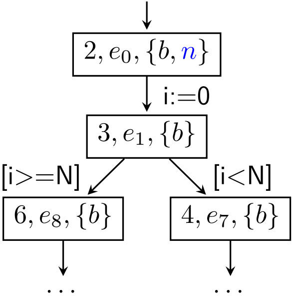


Loop Abstraction with CEGAR: Example 2



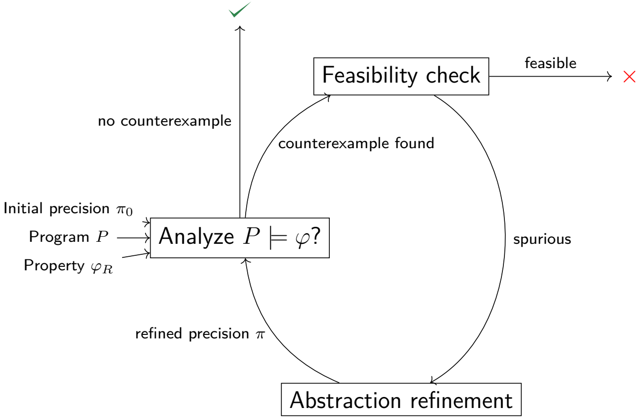






CEGAR: Feasibility of Counterexamples

- 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:
  $\mathtt{refine} : (\text{reached}, \mathtt{waitlist}) \mapsto (\text{reached}', \mathtt{waitlist}')$
  $\text{reached}, \mathtt{waitlist} \subseteq L \times E \times \Pi$
- $\Rightarrow$ We can chain our strategy precision refinement $\mathtt{refine}_{\mathbb{S}}$ with the refinement $\mathtt{refine}_{\mathbb{W}}$ of the wrapped analysis:
  $\mathtt{refine} = \mathtt{refine}_{\mathbb{S}} \circ \mathtt{refine}_{\mathbb{W}}$

Accessibility of Loop Abstractions via Patches

- We provide loop abstractions as patches
- We also output the abstracted version of the program in case we found a proof
- Can be used independently by other tools

```
--- 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
- Easily extensible with new abstraction strategies
- Loop abstractions are made available via patches
- Implemented in the CPAchecker framework,
  cf. supplementary webpage for how to use:
  https://www.sosy-lab.org/research/loop-abstra

Paper PDF:

References I

[1] 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), `http://frama-c.com/download/acsl.pdf`

[2] 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

[3] 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

[4] 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, `https://doi.org/10.1109/APSEC.2012.97`

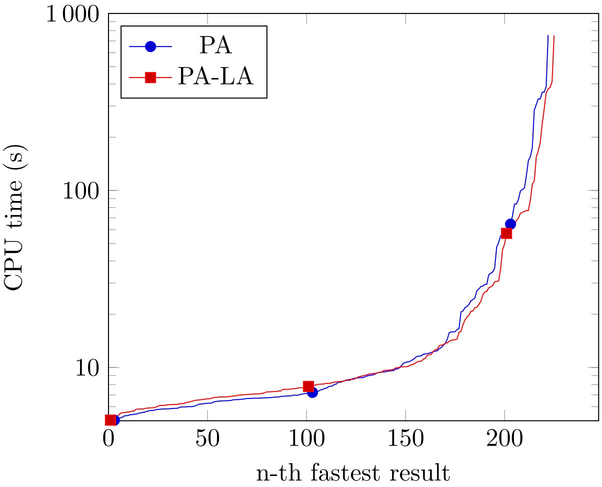
[5] Kumar, S., Sanyal, A., Venkatesh, R., Shah, P.: Property checking array programs using loop shrinking. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on 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, `https://doi.org/10.1007/978-3-319-89960-2_12`

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?

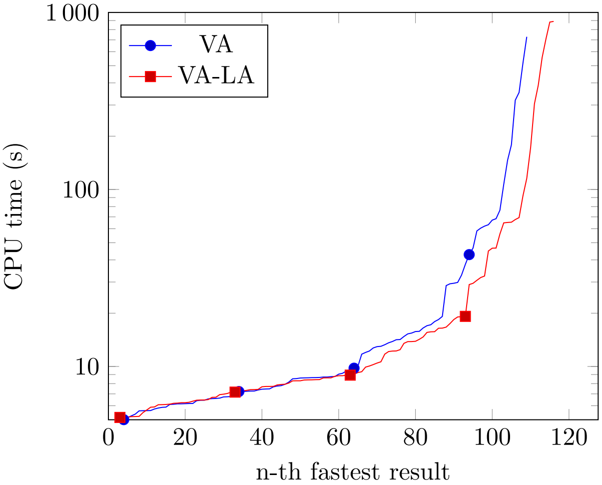
Results for Predicate Abstraction

- Only slightly more tasks solved with loop abstraction
- In many cases, predicate abstraction is already able to proof the program correct
- Overhead is small (as expected)



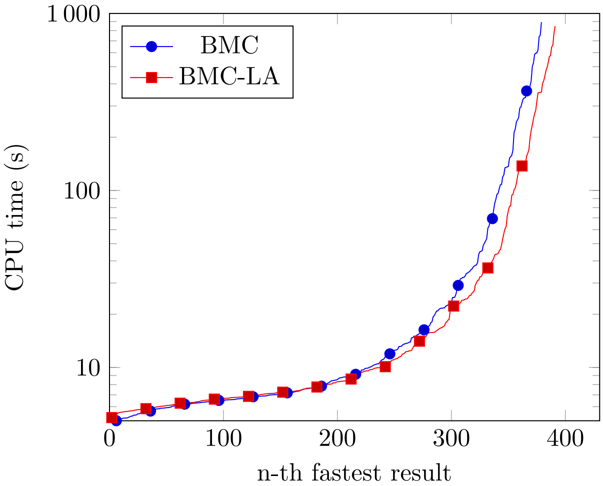
Results for Value Analysis

- Value analysis performs constant propagation
- Less likely to proof program correct on its own
- ⇒ loop abstraction can help to find proofs



Results for Bounded Model Checking

- BMC solves more tasks in general
- effect of loop abstraction comparable to results for value analysis



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 [5]
- 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

```
/*@ requires 0<=n<65536 && *res==0;
 *@ assigns *res;
 *@ ensures *res == n*(n+1)/2; */
void sum(int n, int *res) {
  while (n>0) {*res+=n;n--;}
}
void main() {
  int i = 0;


  sum(1000,&i);

  assert(i==500*1001);
}
```

```
/*@ requires 0<=n<65536 && *res==0;
 *@ assigns *res;
 *@ ensures *res == n*(n+1)/2; */
void sum(int n, int *res) {
  while (n>0) {*res+=n;n--;}
}
void main() {
  int i = 0;
  assert(0<=1000 && 1000<65536);
  assert(i==0);
  havoc(i);
  assume(i==1000*(1000+1)/2);
  assert(i==500500);
}
```

- 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