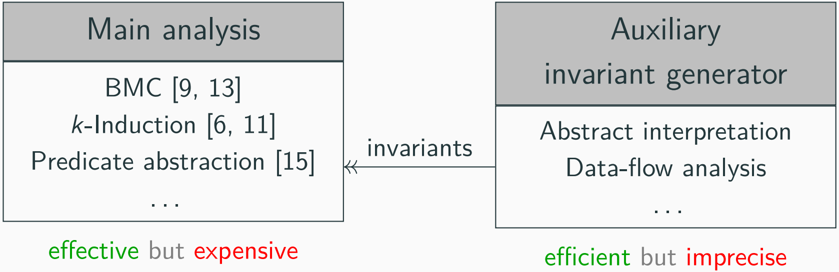
Augmenting Interpolation-Based Model Checking with Auxiliary Invariants

Dirk Beyer, **Po-Chun Chien**, and Nian-Ze Lee

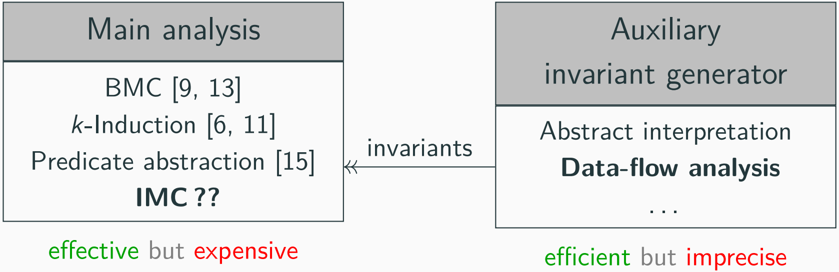


SPIN 2024 @ Luxembourg
LMU Munich, Germany

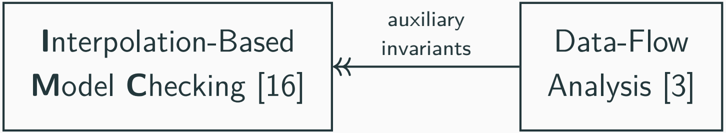
Cooperative Verification via Invariant Injection



Cooperative Verification via Invariant Injection

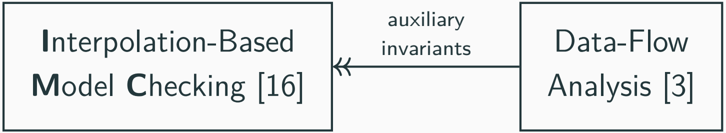


Highlights



- Novelty: 1st to combine IMC with data-flow analysis

Highlights



- Novelty: 1st to combine IMC with data-flow analysis
- In our evaluation, augmented IMC
  - was faster and more effective than plain IMC
  - tackled tasks unsolvable by state-of-the-art verifiers

Example Program

```
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

Example Program

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Reachable states (x, j) at loop head:
  - $(0, 0)$

Example Program

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Reachable states (x, j) at loop head:
  - $(0, 0)$
  - $(2, 1)$

Example Program

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Reachable states (x, j) at loop head:
  - $(0, 0)$
  - $(2, 1)$
  - $(4, 0)$

Example Program

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Reachable states (x, j) at loop head:
  - $(0, 0)$
  - $(2, 1)$
  - $(4, 0)$
  - $(6, 1)$
  - ...

Example Program

```
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Reachable states (x, j) at loop head:
  - $(0, 0)$
  - $(2, 1)$
  - $(4, 0)$
  - $(6, 1)$
  - ...
- ERROR unreachable

Example: Plain IMC

```
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Candidate fixed-point *fp* at loop head: $x\%2 = 0$

Example: Plain IMC

```
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Candidate fixed-point *fp* at loop head: $x\%2 = 0$
- CEX: $x = 0 \land j = 3$

Example: Plain IMC

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Candidate fixed-point *fp* at loop head: $x \% 2 = 0$
- CEX: $x = 0 \land j = 3$

$$\downarrow \text{ 1 loop iteration}$$

$$x = 3 \land j = 4$$

Example: Plain IMC

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Candidate fixed-point *fp* at loop head: $x\%2 = 0$
- CEX: $x = 0 \wedge j = 3$

$$\downarrow \text{ 1 loop iteration}$$

$$x = 3 \wedge j = 4$$

Example: Plain IMC

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Candidate fixed-point *fp* at loop head: $x\%2 = 0$
- CEX: $x = 0 \land j = 3$

  $\downarrow$ 1 loop iteration

  $x = 3 \land j = 4$
- *fp* is non-inductive

  $\rightarrow$ needs refinement!

Example: IMC with Auxililiary Invariants

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Candidate fixed-points at loop head produced by

  IMC: $x\%2 = 0$

Example: IMC with Auxililiary Invariants

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Candidate fixed-points at loop head produced by

  IMC: $x\%2 = 0$

  Data-flow: $0 \leq j \leq 1$

Example: IMC with Auxililiary Invariants

```
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Candidate fixed-points at loop head produced by

  IMC: $x \% 2 = 0$

  Data-flow: $0 \leq j \leq 1$

- $x \% 2 = 0 \land 0 \leq j \leq 1$ is an inductive (and safe) invariant!

Agenda

1. Background: IMC and data-flow analysis
2. Augmenting IMC with auxillary invariants
3. Experimental evaluation

Agenda

1. Background: IMC and data-flow analysis
2. Augmenting IMC with auxillary invariants
3. Experimental evaluation

Interpolation and SAT-Based Model Checking

- K. L. McMillan, CAV 2003 [16]
- Interpolation-based model checking (IMC)
  - Originally designed for finite-state transition systems
  - Compute fixed points by interpolating unsatisfiable BMC queries

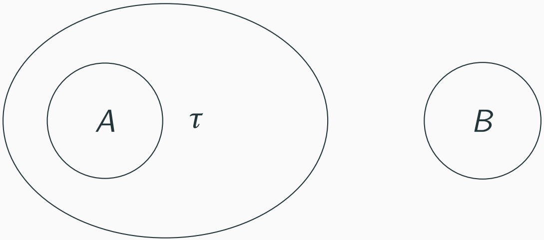
Interpolation and SAT-Based Model Checking

- K. L. McMillan, CAV 2003 [16]
- Interpolation-based model checking (IMC)
  - Originally designed for finite-state transition systems
  - Compute fixed points by interpolating unsatisfiable BMC queries
- State of the art for hardware verification
- Recently adopted for verifying software programs[1]

[1] *Interpolation and SAT-Based Model Checking Revisited: Adoption to Software Verification* (to appear in JAR [7])

Craig Interpolation

- If $A(X,Y) \wedge B(Y,Z)$ is UNSAT: interpolant $\tau(Y)$
  - $A(X,Y) \Rightarrow \tau(Y)$ is valid
  - $\tau(Y) \wedge B(Y,Z)$ is UNSAT



BMC Stage of IMC

- State-transition system: $Init(s), T(s, s')$
- Safety property: $P(s)$

BMC Stage of IMC

- State-transition system: $Init(s), T(s, s')$
- Safety property: $P(s)$
- BMC query with unrolling bound k:
  $Init(s_0)\,T(s_0, s_1)\,T(s_1, s_2) \ldots T(s_{k-1}, s_k)(\neg P(s_1) \vee \ldots \vee \neg P(s_k))$

BMC Stage of IMC

- State-transition system: $Init(s), T(s, s')$
- Safety property: $P(s)$
- BMC query with unrolling bound k:
  $Init(s_0)\,T(s_0, s_1)\,T(s_1, s_2) \ldots T(s_{k-1}, s_k)(\neg P(s_1) \vee \ldots \vee \neg P(s_k))$
  - if SAT, counterexample found
  - if UNSAT, enter *interpolation stage*

Interpolation Stage of IMC

- Construct a fixed point via interpolation

Interpolation Stage of IMC

- Construct a fixed point via interpolation
- $\mathit{Init}(s_0)\, T(s_0, s_1)\ T(s_1, s_2) \dots T(s_{k-1}, s_k)(\neg P(s_1) \lor \dots \lor \neg P(s_k))$

Interpolation Stage of IMC

- Construct a fixed point via interpolation
- $\underbrace{Init(s_0)\,T(s_0,s_1)}_{A_0(s_0,s_1)}\,\underbrace{T(s_1,s_2)\ldots T(s_{k-1},s_k)(\neg P(s_1)\vee\ldots\vee\neg P(s_k))}_{B(s_1,s_2,\ldots,s_k)}$
  - Interpolant $\tau_1(s_1)$: 1-step safe overapproximation

Interpolation Stage of IMC

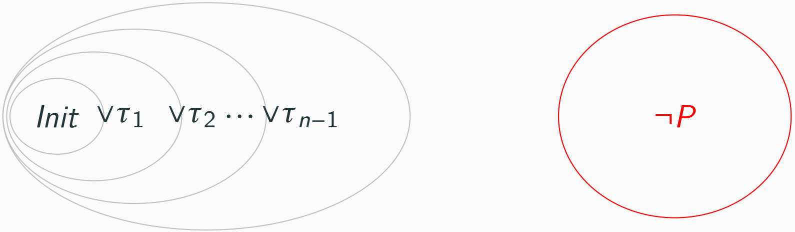
- Construct a fixed point via interpolation
- $\underbrace{Init(s_0)\,T(s_0,s_1)}_{A_0(s_0,s_1)}\;\underbrace{T(s_1,s_2)\ldots T(s_{k-1},s_k)(\neg P(s_1)\lor\ldots\lor\neg P(s_k))}_{B(s_1,s_2,\ldots,s_k)}$
  - Interpolant $\tau_1(s_1)$: 1-step safe overapproximation
- $\underbrace{\tau_1(s_0)\,T(s_0,s_1)}_{A_1(s_0,s_1)}\;\underbrace{T(s_1,s_2)\ldots T(s_{k-1},s_k)(\neg P(s_1)\lor\ldots\lor\neg P(s_k))}_{B(s_1,s_2,\ldots,s_k)}$
  - Interpolant $\tau_2(s_1)$: 2-step safe overapproximation
  - Derive n-step overapproximation τ_n iteratively

Fixed Point

- Repeat until $Init \vee \bigvee \tau_i$ becomes a fixed point

Fixed Point

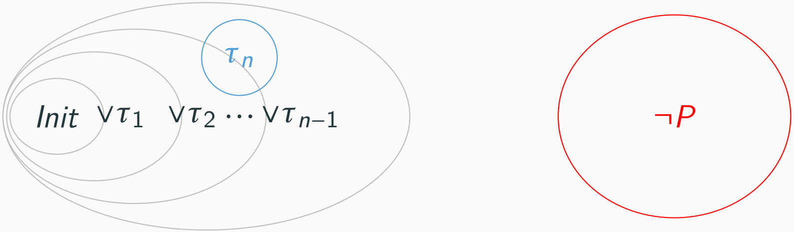
- Repeat until $Init \vee \bigvee \tau_i$ becomes a fixed point



Fixed Point

- Repeat until $Init \lor \bigvee \tau_i$ becomes a fixed point

$\tau_n \Rightarrow Init \lor \bigvee_{i=1}^{n-1} \tau_i$ holds

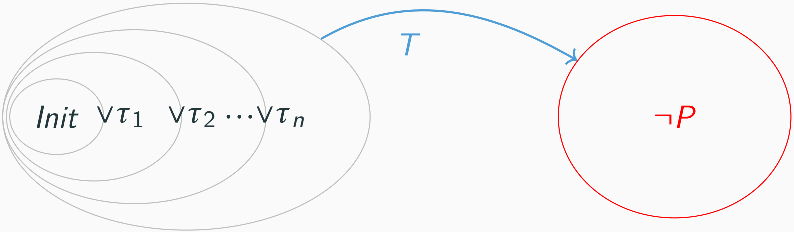


Potentially-Spurious Counterexample

- Increment unrolling bound k if a query becomes satisfiable

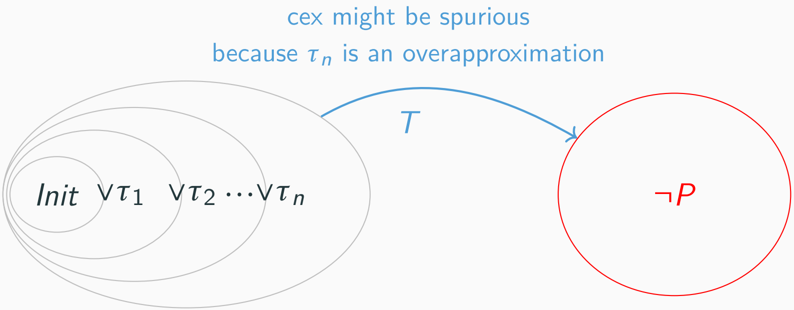
Potentially-Spurious Counterexample

- Increment unrolling bound k if a query becomes satisfiable



Potentially-Spurious Counterexample

- Increment unrolling bound k if a query becomes satisfiable

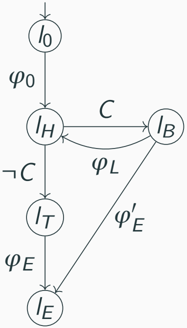


Termination Conditions of IMC

- IMC terminates when
  - A fixed point is reached: return a safety proof
  - $\mathit{Init}(s_0) \land \bigwedge_{i=1}^{k} T(s_{i-1}, s_i) \land (\bigvee_{i=1}^{k} \neg P(s_i))$ is SAT for some k: return a counterexample
- This work improves IMC's capability of constructing proofs

IMC for Software Verification

- Extract formulas by *large-block encoding* [5, 7]

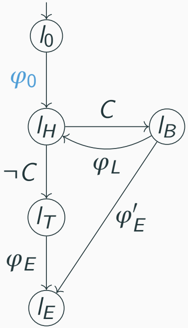


$$(l_0, true)$$
$$\downarrow$$
$$(l_H, \varphi_0|_{i=0})$$
$$\downarrow$$
$$(l_H, (C \land \varphi_L)|_{i=1})$$
$$\vdots$$
$$(l_H, (C \land \varphi_L)|_{i=k})$$
$$\downarrow$$
$$(l_E, ((\neg C \land \varphi_E) \lor (C \land \varphi'_E))|_{i=k})$$

(For multi-loop programs: standard transformation to single loop [1, 12])

IMC for Software Verification

- Extract formulas by *large-block encoding* [5, 7]

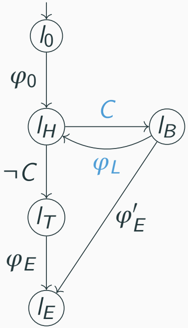


$$(l_0, true)$$
$$\downarrow$$
$$(l_H, \varphi_0|_{i=0}) \ : Init(s_0)$$
$$\downarrow$$
$$(l_H, (C \wedge \varphi_L)|_{i=1})$$
$$\vdots$$
$$(l_H, (C \wedge \varphi_L)|_{i=k})$$
$$\downarrow$$
$$(l_E, ((\neg C \wedge \varphi_E) \vee (C \wedge \varphi'_E))|_{i=k})$$

(For multi-loop programs: standard transformation to single loop [1, 12])

IMC for Software Verification

- Extract formulas by *large-block encoding* [5, 7]

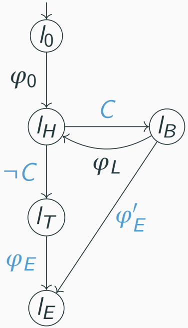


$$(l_0, true)$$
$$\downarrow$$
$$(l_H, \varphi_0|_{i=0}) \;:\; Init(s_0)$$
$$\downarrow$$
$$(l_H, (C \land \varphi_L)|_{i=1}) \;:\; T(s_0, s_1)$$
$$\vdots$$
$$(l_H, (C \land \varphi_L)|_{i=k}) \;:\; T(s_{k-1}, s_k)$$
$$\downarrow$$
$$(l_E, ((\neg C \land \varphi_E) \lor (C \land \varphi'_E))|_{i=k})$$

(For multi-loop programs: standard transformation to single loop [1, 12])

IMC for Software Verification

- Extract formulas by *large-block encoding* [5, 7]



$$(l_0, true)$$
$$\downarrow$$
$$(l_H, \varphi_0|_{i=0}) \; : Init(s_0)$$
$$\downarrow$$
$$(l_H, (C \wedge \varphi_L)|_{i=1}) \; : T(s_0, s_1)$$
$$\vdots$$
$$(l_H, (C \wedge \varphi_L)|_{i=k}) \; : T(s_{k-1}, s_k)$$
$$\downarrow$$
$$(l_E, ((\neg C \wedge \varphi_E) \vee (C \wedge \varphi'_E))|_{i=k}) \; : \neg P(s_k)$$

(For multi-loop programs: standard transformation to single loop [1, 12])

Auxiliary Invariant Generator

- Continuously-refining data-flow analysis (DF) based on intervals [3]
- Produce **inductive** invariants
- Invariants are expressions over intervals
  - e.g. $(0 \le j \le 1) \land (x < 5 \lor x > 7)$
- Invariant injection denoted as $\leftarrow\!\circlearrowleft\!-$DF

Agenda

Strengthen Interpolants with Auxililiary Invariants

- Given an **inductive** invariant Inv, interpolant τ_i can be strengthened by

$$\tau'_i \leftarrow \tau_i \wedge \mathit{Inv}$$

Strengthen Interpolants with Auxililiary Invariants

- Given an **inductive** invariant Inv, interpolant τ_i can be strengthened by

$$\tau'_i \leftarrow \tau_i \land Inv$$

- τ'_i is a valid interpolant for IMC

Strengthen Interpolants with Auxililiary Invariants

- Given an **inductive** invariant Inv, interpolant τ_i can be strengthened by

$$\tau'_i \leftarrow \tau_i \land Inv$$

- τ'_i is a valid interpolant for IMC
- Inv helps remove some **unreachable** states in τ_i
- $\tau'(s_0) \land \bigwedge_{i=1}^{k} T(s_{i-1}, s_i) \land (\bigvee_{i=1}^{k} \neg P(s_i))$ is more likely to remain UNSAT

Alternative Ways to Utilize Auxililiary Invariants

- Injecting auxiliary invariants into
  - Fixed-point check: $\boldsymbol{Inv} \wedge \tau_n \Rightarrow Init \vee \bigvee_{i=1}^{n-1} \tau_i$
  - Safety property: $P' \leftarrow P \wedge \boldsymbol{Inv}$

Alternative Ways to Utilize Auxililiary Invariants

- Injecting auxiliary invariants into
  - Fixed-point check: $\boldsymbol{Inv} \wedge \tau_n \Rightarrow Init \vee \bigvee_{i=1}^{n-1} \tau_i$
  - Safety property: $P' \leftarrow P \wedge \boldsymbol{Inv}$
- Not as effective as strengthening interpolants

Agenda

1. Background: IMC and data-flow analysis
2. Augmenting IMC with auxillary invariants
3. Experimental evaluation

Evaluation

- We conducted experiments to answer the following research questions:
  - **RQ1**: Can auxiliary invariants help improve IMC?
  - **RQ2**: Is the augmented IMC competitive?

Evaluation

- We conducted experiments to answer the following research questions:
  - **RQ1**: Can auxiliary invariants help improve IMC?
  - **RQ2**: Is the augmented IMC competitive?
- Reproduction package [4] available and reusable!

Implementation and Configurations in CPAchecker

- CPAchecker[2]: revision 42901 of branch *imc-with-invariants*
- Interpolants computed by MathSAT5 [10] (theory: QF_ABVFPUF)

[2] https://cpachecker.sosy-lab.org/

Implementation and Configurations in CPAchecker

- CPACHECKER[2]: revision 42901 of branch *imc-with-invariants*
- Interpolants computed by MATHSAT5 [10] (theory: QF_ABVFPUF)
- Compared SMT-based algorithms
  - IMC [7] vs. IMC←⟲-DF
  - KI←⟲-DF [6], predicate abstraction [14], IMPACT [17]

[2] https://cpachecker.sosy-lab.org/

Benchmark Tools and Tasks

- Compared software verifiers (from SV-COMP 2022 [2])
  - 2LS [8]: *k*-induction boosted by auxiliary invariants
  - SYMBIOTIC [18]: the overall winner of SV-COMP 2022

Benchmark Tools and Tasks

- Compared software verifiers (from SV-COMP 2022 [2])
  - 2LS [8]: *k*-induction boosted by auxiliary invariants
  - Symbiotic [18]: the overall winner of SV-COMP 2022
- Benchmark set: *ReachSafety* tasks of SV-COMP 2022 [2]
  - No property violation (i.e., safe)
  - Focus on single-loop programs
  - Eliminate easy ones solvable by CPAchecker's BMC within 900 s
  - 1623 after filtering
  - DF can produce non-trivial invariants on 870 tasks

Experimental Setup

- Environment
  - OS: Ubuntu 22.04 (64 bit)
  - Machine: 3.4 GHz CPU (8 cores) and 33 GB of RAM
- Each task is limited to
  - 4 CPU cores
  - 900 s of CPU time (max 150 s for DF)
  - 15 GB of RAM

  (reliable resource management by BenchExec[3])

[3] `https://github.com/sosy-lab/benchexec`

RQ1: Can auxiliary invariants help improve IMC?

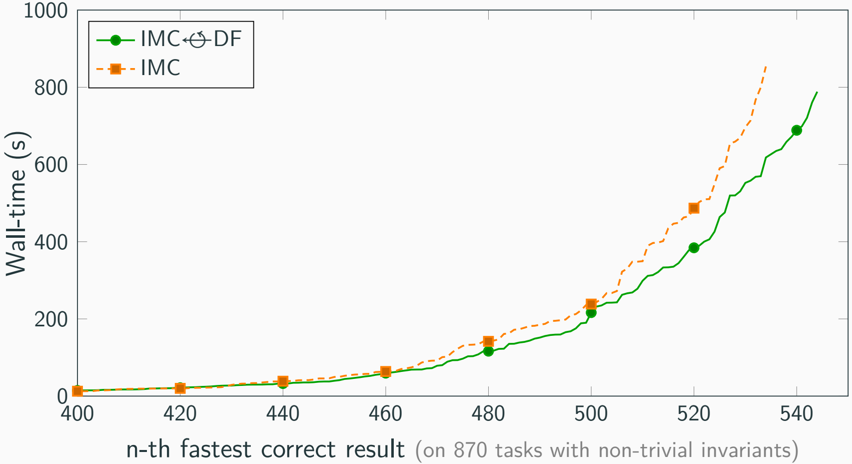
Improved Effectiveness

Task	IMC (timeout)			IMC⟲DF (solved)		
	#unroll	#itp	wall-time	#unroll	#itp	wall-time
Problem03_label51	10	57	878	5	11	24.3
benchmark37_conj	317	316	892	1	2	1.83
s3_srvr_1a.BV.c.cil	64	441	885	5	13	6.26

(time unit: s; hand-picked tasks with significant improvement)

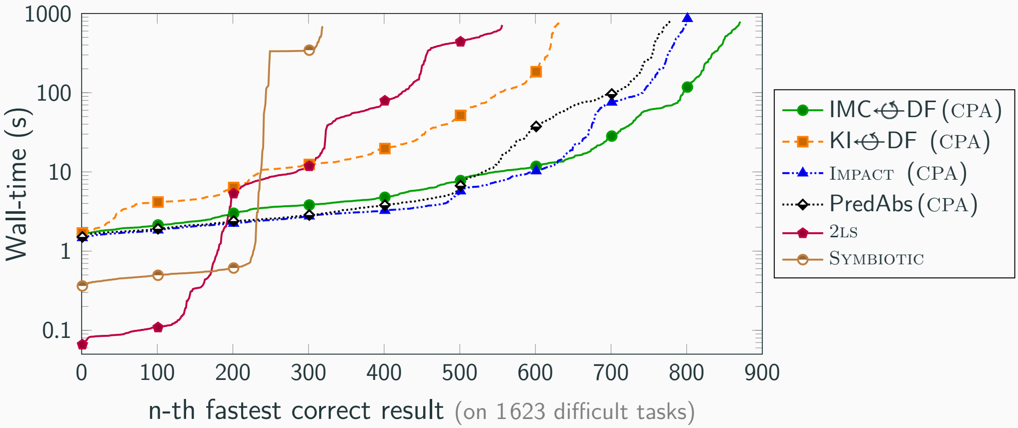
- IMC⟲DF solved **23** tasks where plain IMC ran into timeouts
  (on tasks with non-trivial auxiliary invariants)

Comparing Elapsed Wall-Time



RQ2: Is the augmented IMC competitive?

Comparsion with Others



Answers to RQs

- **RQ1**: Can auxiliary invariants help improve IMC?
- **RQ2**: Is the augmented IMC competitive?

Answers to RQs

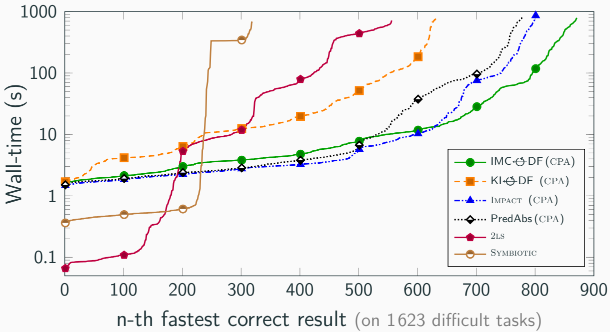
- **RQ1**: Can auxiliary invariants help improve IMC?
  Yes, effectiveness and wall-time efficiency are improved
- **RQ2**: Is the augmented IMC competitive?

Answers to RQs

- **RQ1**: Can auxiliary invariants help improve IMC?
  Yes, effectiveness and wall-time efficiency are improved
- **RQ2**: Is the augmented IMC competitive?
  Yes, more proofs compared to other verification algorithms and tools

Conclusion

- Augment IMC [16] via invariant injection
- Open-source implementation in CPACHECKER



www.sosy-lab.org/research/imc-df/

References i

[1] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986). https://www.worldcat.org/isbn/978-0-201-10088-4

[2] Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20

[3] Beyer, D., Chien, P.C., Lee, N.Z.: CPA-DF: A tool for configurable interval analysis to boost program verification. In: Proc. ASE. pp. 2050–2053. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00213

[4] Beyer, D., Chien, P.C., Lee, N.Z.: Reproduction package for SPIN 2024 submission 'Augmenting interpolation-based model checking with auxiliary invariants'. Zenodo (2024). https://doi.org/10.5281/zenodo.10548594

[5] Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proc. FMCAD. pp. 25–32. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351147

References ii

[6] Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42

[7] Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. J. Autom. Reasoning (2024), accepted, preprint available via https://doi.org/10.48550/arXiv.2208.05046

[8] Brain, M., Joshi, S., Kröning, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Proc. SAS. pp. 145–161. LNCS 9291, Springer (2015). https://doi.org/10.1007/978-3-662-48288-9_9

[9] Cheng, X., Hsiao, M.S.: Simulation-directed invariant mining for software verification. In: Proc. DATE. pp. 682–687. ACM (2008). https://doi.org/10.1109/DATE.2008.4484757

[10] Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS. pp. 93–107. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7

References iii

[11] Donaldson, A.F., Haller, L., Kröning, D.: Strengthening induction-based race checking with lightweight static analysis. In: Proc. VMCAI. pp. 169–183. LNCS 6538, Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_13

[12] Donaldson, A.F., Kröning, D., Rümmer, P.: Automatic analysis of DMA races using model checking and *k*-induction. FMSD **39**(1), 83–113 (2011). https://doi.org/10.1007/s10703-011-0124-2

[13] Ganai, M.K., Gupta, A.: Accelerating high-level bounded model checking. In: Proc. ICCAD. pp. 794–801. ACM (2006). https://doi.org/10.1145/1233501.1233664

[14] Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021

[15] Jain, H., Ivancic, F., Gupta, A., Shlyakhter, I., Wang, C.: Using statically computed invariants inside the predicate abstraction and refinement loop. In: Proc. CAV. pp. 137–151. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_15

References iv

[16] McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV. pp. 1–13. LNCS 2725, Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1

[17] McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV. pp. 123–136. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_14

[18] Slabý, J., Strejček, J., Trtík, M.: Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution. In: Proc. FMICS. pp. 207–221. LNCS 7437, Springer (2012). https://doi.org/10.1007/978-3-642-32469-7_14

Example Revisited: Collecting Formulas

```
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Large-block encoding [5]
- $s \leftarrow \{x, j\}$

Example Revisited: Collecting Formulas

I

```
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Large-block encoding [5]
- $s \leftarrow \{x, j\}$
- $Init(s) \leftarrow (x = 0) \land (j = 0)$

Example Revisited: Collecting Formulas

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Large-block encoding [5]
- $s \leftarrow \{x, j\}$
- $Init(s) \leftarrow (x = 0) \land (j = 0)$
- $T(s, s') \leftarrow (x_1 = x + 2)$
  $\land (j = 3 \Rightarrow x' = x_1 + 1)$
  $\land (j \neq 3 \Rightarrow x' = x_1)$
  $\land (j_1 = j + 1)$
  $\land (j_1 = 2 \Rightarrow j' = 0)$
  $\land (j_1 \neq 2 \Rightarrow j' = j_1)$

Example Revisited: Collecting Formulas

```
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

- Large-block encoding [5]
- $s \leftarrow \{x, j\}$
- $Init(s) \leftarrow (x = 0) \wedge (j = 0)$
- $T(s, s') \leftarrow (x_1 = x + 2)$
  $\wedge (j = 3 \Rightarrow x' = x_1 + 1)$
  $\wedge (j \neq 3 \Rightarrow x' = x_1)$
  $\wedge (j_1 = j + 1)$
  $\wedge (j_1 = 2 \Rightarrow j' = 0)$
  $\wedge (j_1 \neq 2 \Rightarrow j' = j_1)$
- $P(s') \leftarrow x' \% 2 = 0$

Example Revisited: Plain IMC

```c
int main(void) {
```

I

```c
  unsigned x = 0;
  unsigned j = 0;
```

T

```c
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
```

P

```c
  if (x % 2) {
    ERROR: return 1;
  }
```

```c
  return 0;
}
```

One loop unrolling ($k = 1$)

- $Init \wedge T \wedge \neg P$ is UNSAT
- $\tau_1 \leftarrow x\%2 = 0$

Example Revisited: Plain IMC

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

(Brackets beside the code: I = lines 2–3, T = lines 4–11, P = lines 12–14)

One loop unrolling ($k = 1$)

- $Init \wedge T \wedge \neg P$ is UNSAT
- $\tau_1 \leftarrow x\%2 = 0$
- $\tau_1 \wedge T \wedge \neg P$ is SAT
  (spurious cex: $x = 0 \wedge j = 3$)

Example Revisited: Plain IMC

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

One loop unrolling ($k = 1$)

- $Init \land T \land \neg P$ is UNSAT
- $\tau_1 \leftarrow x \% 2 = 0$
- $\tau_1 \land T \land \neg P$ is SAT
  (spurious cex: $x = 0 \land j = 3$)
- Increment k

Example Revisited: Augmented IMC

```
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

One loop unrolling $(k = 1)$

- $Inv \leftarrow 0 \leq j \leq 1$

Example Revisited: Augmented IMC

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

One loop unrolling $(k = 1)$

- $Inv \leftarrow 0 \leq j \leq 1$
- $Init \wedge T \wedge \neg P$ is UNSAT
- $\tau_1 \leftarrow x\%2 = 0; \quad \boldsymbol{\tau'_1 \leftarrow \tau_1 \wedge Inv}$
- $\tau'_1 \wedge T \wedge \neg P$ is UNSAT

Example Revisited: Augmented IMC

```
      int main(void) {
I       unsigned x = 0;
        unsigned j = 0;
        while (nondet()) {
          x += 2;
          if (j == 3)
T           x += 1;
          j += 1;
          if (j == 2)
            j = 0;
        }
        if (x % 2) {
P         ERROR: return 1;
        }
        return 0;
      }
```

One loop unrolling $(k = 1)$

- $Inv \leftarrow 0 \leq j \leq 1$
- $Init \wedge T \wedge \neg P$ is UNSAT
- $\tau_1 \leftarrow x\%2 = 0$; $\boldsymbol{\tau'_1 \leftarrow \tau_1 \wedge Inv}$
- $\tau'_1 \wedge T \wedge \neg P$ is UNSAT
- $\tau_2 \leftarrow x\%2 = 0$; $\boldsymbol{\tau'_2 \leftarrow \tau_2 \wedge Inv}$

Example Revisited: Augmented IMC

```c
int main(void) {
  unsigned x = 0;
  unsigned j = 0;
  while (nondet()) {
    x += 2;
    if (j == 3)
      x += 1;
    j += 1;
    if (j == 2)
      j = 0;
  }
  if (x % 2) {
    ERROR: return 1;
  }
  return 0;
}
```

One loop unrolling $(k = 1)$

- $Inv \leftarrow 0 \leq j \leq 1$
- $Init \wedge T \wedge \neg P$ is UNSAT
- $\tau_1 \leftarrow x\%2 = 0; \quad \boldsymbol{\tau'_1 \leftarrow \tau_1 \wedge Inv}$
- $\tau'_1 \wedge T \wedge \neg P$ is UNSAT
- $\tau_2 \leftarrow x\%2 = 0; \quad \boldsymbol{\tau'_2 \leftarrow \tau_2 \wedge Inv}$
- $\tau'_2 \Rightarrow (I \vee \tau'_1)$ holds: fixed point!

Critical Reflection

- Auxiliary invariants brought *net improvement* to IMC
- However...

Critical Reflection

- Auxiliary invariants brought *net improvement* to IMC
- However...
  - The improvement was not remarkable
  - Some tasks became unsolvable with added invariants
- Reasons:
  - Invariant generator consumed additional CPU time
  - Interpolation queries became more difficult for the solver