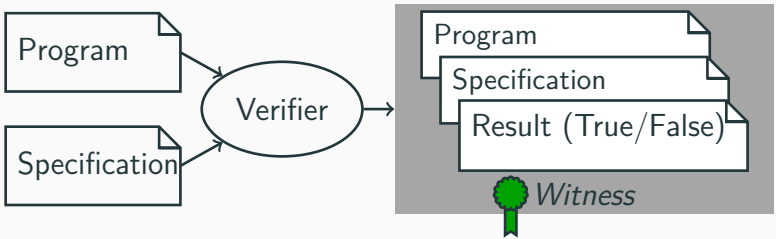
Distributed Summary Synthesis: A Divide-and-Conquer Approach for Distributed Program Analysis

Dirk Beyer

LMU Munich, Germany

Dagstuhl Seminar 25421
2025-10-15

Automatic Software Verification



Motivation

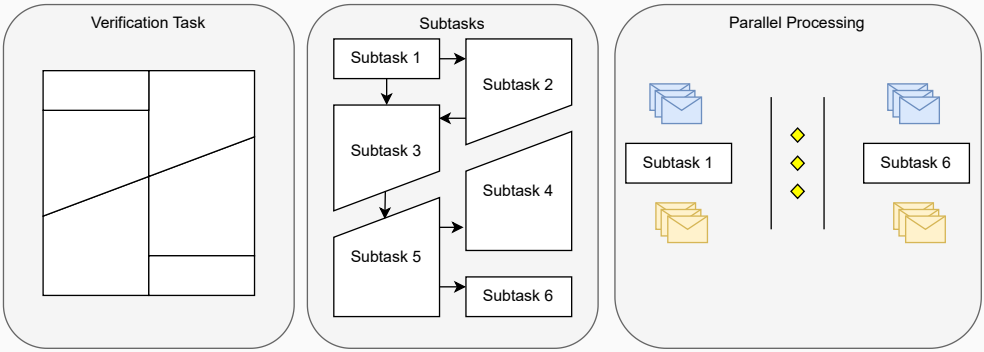
- We need low response time (enable MC in continuous integration).
- We are looking for a massively parallel approach.
- Solution: Decomposition into blocks,
  construct block-wise contracts using backend verifiers
- Goal: Scalable and Distributed Software Verification

Based on [6]:

Dirk Beyer, Matthias Kettl, Thomas Lemberger:

Decomposing Software Verification using Distributed Summary Synthesis

Proc. ACM on Software Engineering, Volume 1, Issue FSE, 2024.
https://doi.org/10.1145/3660766



Overview of the *DSS* approach

Decomposition

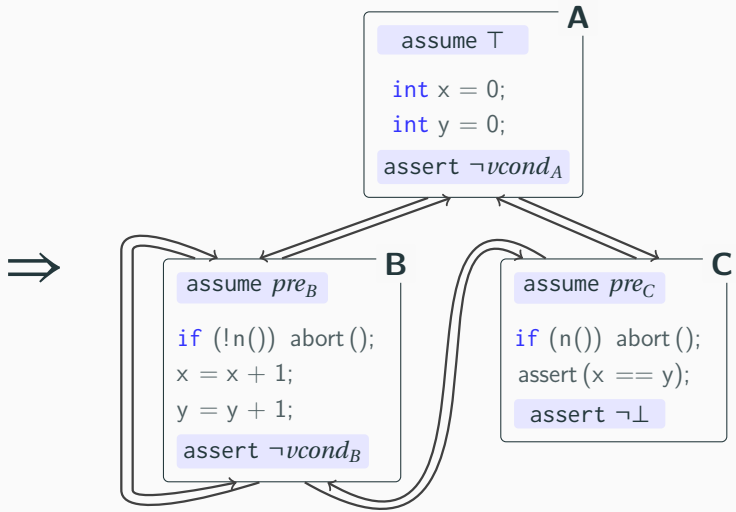
Requirements for our current decomposition into blocks:

- Each block has exactly one entry and one exit location.
- Loops should be reflected as loops in the block graph.
- Blocks should be as large as possible.
- Blocks not bound to functions.

Approach: We decompose the CFA similar to large-block encoding [3].

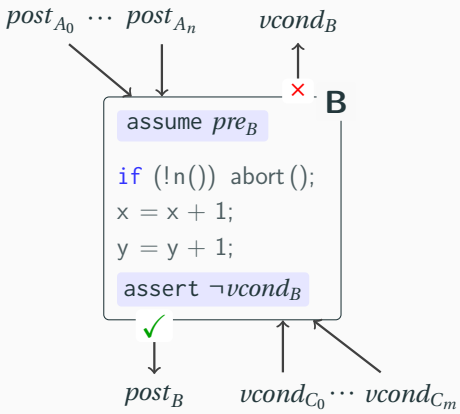
Decomposition

```
1 int main() {
2   int x = 0;
3   int y = 0;
4   while (n()) {
5     x = x + 1;
6     y = y + 1;
7   }
8   assert(x == y);
9 }
```



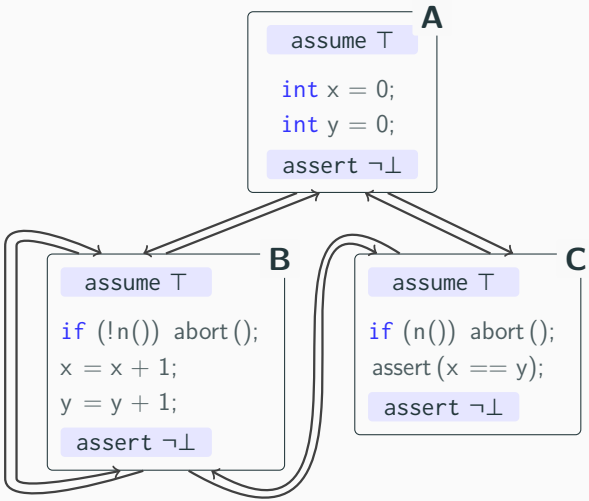
Workers

- Preconditions describe good entry states of a block (over-approximating).
- Violation condition needs to be refuted to prove a program safe (precise or over-approximation).
- Preconditions are refined until all violation conditions are refuted or at least one is confirmed.



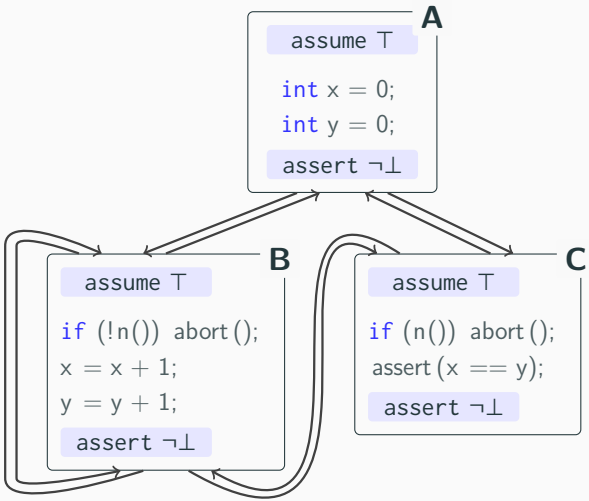
DSS Example

it	**Block**	*pre*	*vcond*	**Result**



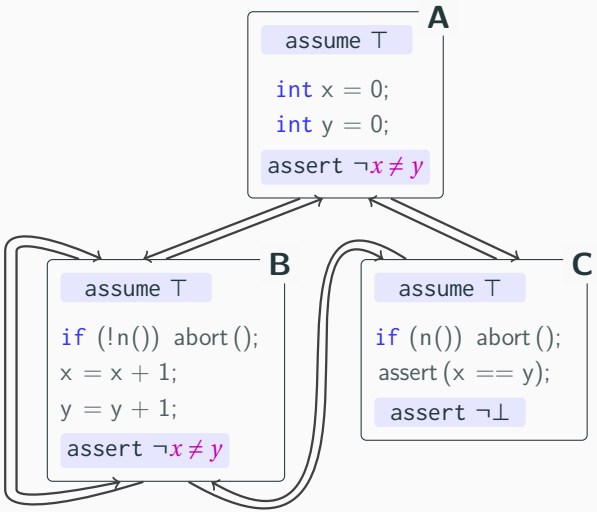
DSS Example

it	Block	*pre*	*vcond*	Result
1	A	$\top$	$\bot$	$\downarrow\boxtimes$: $\top$
1	B	$\top$	$\bot$	$\downarrow\boxtimes$: $\top$
1	C	$\bot$	$\top$	$\uparrow\boxtimes$: $x \neq y$



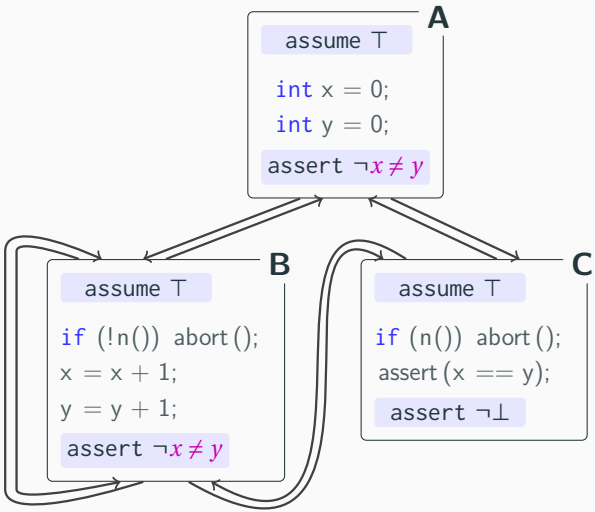
DSS Example

it	Block	*pre*	*vcond*	Result
1	A	$\top$	$\bot$	$\downarrow\boxtimes: \top$
1	B	$\top$	$\bot$	$\downarrow\boxtimes: \top$
1	C	$\bot$	$\top$	$\uparrow\boxtimes: x \neq y$



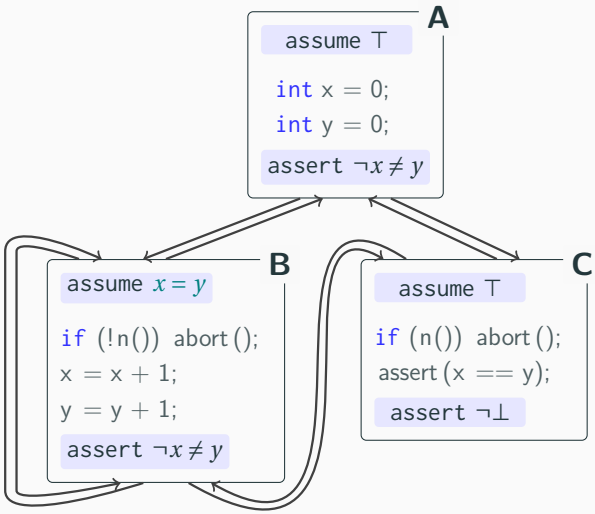
DSS Example

it	**Block**	*pre*	*vcond*	**Result**
1	A	$\top$	$\bot$	$\downarrow\boxtimes$: $\top$
1	B	$\top$	$\bot$	$\downarrow\boxtimes$: $\top$
1	C	$\bot$	$\top$	$\uparrow\boxtimes$: $x \neq y$
2	A	$\top$	$x \neq y$	$\downarrow\boxtimes$: $x = y$
2	B	$\top$	$x \neq y$	$\uparrow\boxtimes$: $x \neq y$



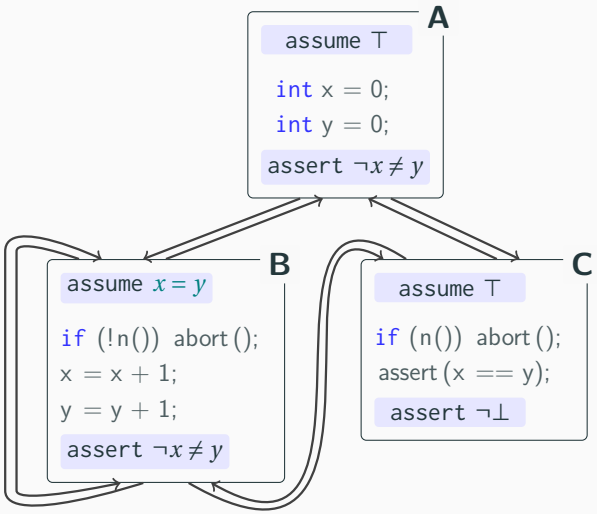
DSS Example

it	**Block**	*pre*	*vcond*	**Result**
1	A	$\top$	$\bot$	$\downarrow\boxtimes$: $\top$
1	B	$\top$	$\bot$	$\downarrow\boxtimes$: $\top$
1	C	$\bot$	$\top$	$\uparrow\boxtimes$: $x \neq y$
2	A	$\top$	$x \neq y$	$\downarrow\boxtimes$: $x = y$
2	B	$\top$	$x \neq y$	$\uparrow\boxtimes$: $x \neq y$



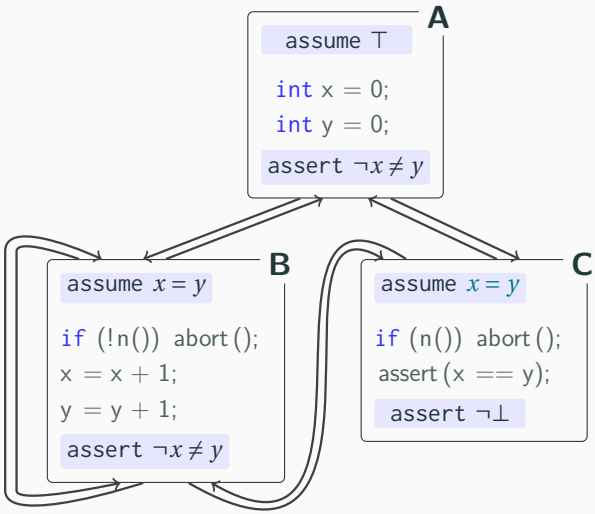
DSS Example

it	Block	*pre*	*vcond*	Result
1	A	$\top$	$\bot$	$\downarrow\boxtimes$: $\top$
1	B	$\top$	$\bot$	$\downarrow\boxtimes$: $\top$
1	C	$\bot$	$\top$	$\uparrow\boxtimes$: $x \neq y$
2	A	$\top$	$x \neq y$	$\downarrow\boxtimes$: $x = y$
2	B	$\top$	$x \neq y$	$\uparrow\boxtimes$: $x \neq y$
3	B	$x = y$	$x \neq y$	$\downarrow\boxtimes$: $x = y$



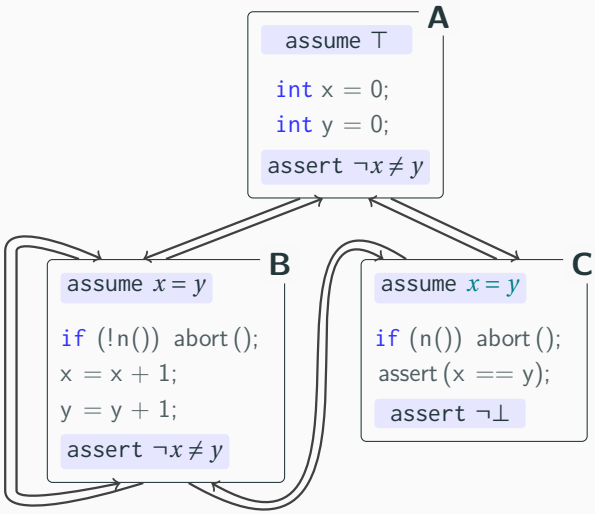
DSS Example

it	Block	*pre*	*vcond*	Result
1	A	$\top$	$\bot$	$\downarrow\boxtimes$: $\top$
1	B	$\top$	$\bot$	$\downarrow\boxtimes$: $\top$
1	C	$\bot$	$\top$	$\uparrow\boxtimes$: $x \neq y$
2	A	$\top$	$x \neq y$	$\downarrow\boxtimes$: $x = y$
2	B	$\top$	$x \neq y$	$\uparrow\boxtimes$: $x \neq y$
3	B	$x = y$	$x \neq y$	$\downarrow\boxtimes$: $x = y$



DSS Example

it	Block	*pre*	*vcond*	Result
1	A	$\top$	$\bot$	$\downarrow\boxtimes$: $\top$
1	B	$\top$	$\bot$	$\downarrow\boxtimes$: $\top$
1	C	$\bot$	$\top$	$\uparrow\boxtimes$: $x \neq y$
2	A	$\top$	$x \neq y$	$\downarrow\boxtimes$: $x = y$
2	B	$\top$	$x \neq y$	$\uparrow\boxtimes$: $x \neq y$
3	B	$x = y$	$x \neq y$	$\downarrow\boxtimes$: $x = y$
4	C	$x = y$	$\bot$	$\downarrow\boxtimes$: $\top$



Details on Information Exchange

- Subtasks are identified via their program location
- Message consists of
  relevant program location + (postcondition | violation condition)
- Postcondition:
  - Symbolic summary at block end (e.g., Craig interpolant)
  - Communicated in SMT-LIB
- Violation conditions:
  - Precise path formula
  - Communicated in SMT-LIB

Related Approaches

Existing approaches have limitations that distributed summary synthesis solves, most importantly the potential to scale to many nodes:

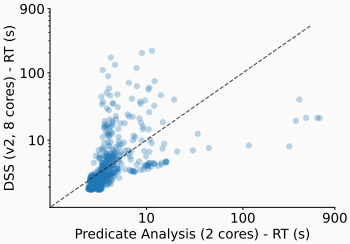
- INFER [7, 8] scales well but reports many false alarms.
  ⇒ *DSS* is not restricted to a local context.
- BAM [4] has nested blocks that are not parallelizable.
  ⇒ *DSS* does not wait for predecessor blocks and parallelizes as much as possible.
- HIFROG [1] is bound to function summaries and SMT-based algorithms.
  ⇒ *DSS* is domain-independent and can use flexible block size.

Evaluation: Setup

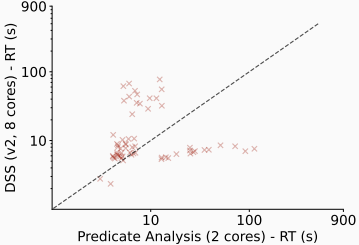
Benchmark Setup:

- We implemented DSS in CPAchecker [5].
- Two abstract domains: predicate abstraction, symbolic execution.
- We evaluate *DSS* on the subcategories *SoftwareSystems* and *ProductLines* of the SV-COMP '25 benchmarks (654 unsafe and 2935 safe verification tasks).
- We use the SV-COMP [2] benchmark setup:
  15 GB RAM, 15 min, on an 8-core Intel Xeon E3-1230 v5 with 3.40 GHz.

Evaluation: Results for Predicate Abstraction



safe tasks, p-value=0.000 14

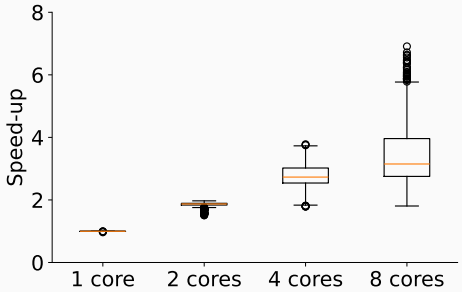


unsafe tasks, p-value=0.96

Response time of predicate abstraction (x-axis) vs. *DSS* (y-axis),
p-values computed using the Wilcoxon signed-rank test for "DSS is faster"

DSS introduces overhead which only pays-off for more complex tasks.
A parallel portfolio can combine the best of both approaches.

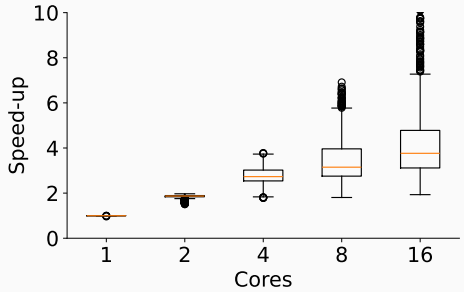
Evaluation: Distribution of Workload



The ratio of the CPU time compared to the response time for 1, 2, 4, and 8 cores.

The workload is distributed effectively to multiple processing units.

Evaluation: Distribution of Workload



The ratio of the CPU time compared to the response time for 1, 2, 4, 8, and 16 cores. For 16 cores, we used an AMD Ryzen 7 PRO 5750G.

The workload is distributed effectively to multiple processing units.

Evaluation: Outperforming Predicate Abstraction

Task	$\text{CPU}_{\mathbb{P}}(s)$	$\text{CPU}_{DSS}(s)$	$\text{RT}_{\mathbb{P}}(s)$	$\mathbf{RT}_{DSS}(s)$	Block size (avg.)	# Threads
minepump-spec4-produ...	640	140	630	21	6.8	70
product-lines/minepu...	61	42	51	8.5	5.9	85
regulator–wm831x-ld...	45	27	34	12	9.7	353
tty–serial–xilinx-...	24	22	15	8.9	15	217
watchdog–w83697hf-w...	24	17	15	7.5	8.0	204

DSS outperforms predicate abstraction on some tasks.
With the help of a parallel portfolio we get the best of both worlds.

Evaluation: DSS with more Resources

Task	$\mathrm{CPU}_{DSS}(s)$	$\mathrm{RT}_{DSS}(s)$
drivers–hid–hid-...	2700	390
minepump-spec1-product49...	2100	570
hid–hid-topseed.ko-ldv...	1700	260
drivers–hwmon–s3c-hwmon...	860	86
stmpe-ts.ko-ldv-main0-seq...	730	140

Increasing the number of cores to 16, the memory to 32 GB and the time limit to 15 min response time allows *DSS* to solve more complex tasks.

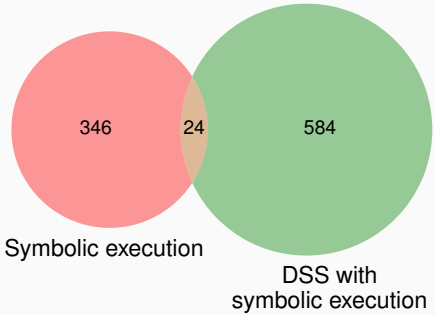
Evaluation: Increasing the Number of Cores

Task	1 core	2 cores	4 cores	8 cores	16 cores	32 cores	64 cores	128 cores
ldv-linux-3.4-simple...	51	19	19	9.6	8.3	8.2	8.7	8.7
ldv-linux-3.4-simple...	630	540	410	340	310	320	300	330
misc–bmp085.ko-ldv-...	390	280	170	160	260	230	160	170

More cores decrease the response time for complex tasks.
However, this may cause more overhead and more messages to be processed.

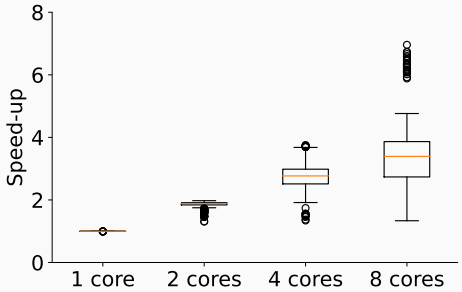
Evaluation: DSS with Symbolic Execution (DSS_{SE})

	SE	DSS_{SE}	Virtual best
Proofs	71	**585**	655
Violations	**299**	23	299
Total	370	608	954



DSS with symbolic execution proves more tasks than symbolic execution alone, but is less efficient at finding counterexamples.

Evaluation: DSS_{SE} Distribution of Workload



The ratio of the CPU time compared to the response time for 1, 2, 4, and 8 cores.

The workload is distributed effectively to multiple processing units.

Evaluation: DSS_{SE} Outperforming Symbolic Execution

Task	$CPU_{SE}(s)$	$CPU_{DSS}(s)$	$RT_{SE}(s)$	$\mathbf{RT}_{DSS}(s)$	Block size (avg.)	# Threads
misc–sgi-xp–xp.ko-...	900	13	890	3.2	18	38
net–phy–et1011c.ko...	900	13	890	3.2	36	28
media–video–cx88–...	900	14	890	3.3	51	28
net–phy–national.k...	900	14	890	3.4	38	29
usb–storage–ums-sd...	900	13	890	3.4	60	16

DSS_{SE} can quickly prove many tasks where symbolic execution times out. This is mostly due to the nature of the decomposition rather than the distribution.

Conclusion

- *DSS* can decompose a verification task into independent smaller tasks.
- *DSS* is domain-independent and supports various block sizes.
- *DSS* effectively distributes the workload to multiple processing units.
- *DSS* is implemented in CPAchecker [5].

Supplementary webpage

References I

[1] Alt, L., Asadi, S., Chockler, H., Even-Mendoza, K., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: HiFrog: SMT-based function summarization for software verification. In: Proc. TACAS. pp. 207–213. LNCS 10206 (2017). https://doi.org/10.1007/978-3-662-54580-5_12

[2] Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3). pp. 299–329. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_15

[3] 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

[4] Beyer, D., Friedberger, K.: Domain-independent interprocedural program analysis using block-abstraction memoization. In: Proc. ESEC/FSE. pp. 50–62. ACM (2020). https://doi.org/10.1145/3368089.3409718

[5] Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16

[6] Beyer, D., Kettl, M., Lemberger, T.: Decomposing software verification using distributed summary synthesis. Proc. ACM Softw. Eng. **1**(FSE) (2024). https://doi.org/10.1145/3660766

References II

[7] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1

[8] Kettl, M., Lemberger, T.: The static analyzer Infer in SV-COMP (competition contribution). In: Proc. TACAS (2). pp. 451–456. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_30