The University of Manchester

EBF 4.2: Black-Box Cooperative Verification for Concurrent Programs

Fatimah Aljaafari, Fedor Shmarov, Edoardo Manino, Rafael Menezes, Lucas Cordeiro

University of Manchester
Department of Computer Science

MANCHESTER
1824
The University of Manchester

Introduction

- **Concurrency** is prevalent in present-day software systems.

  - computer games
  - ticket reservation systems
  - online banking
  - auto-pilots
  - …

- Ensuring the **correctness** and **safety** of **concurrent** programs is crucial
  - Software failures may lead to significant financial losses and affect people’s well-being.

MANCHESTER 1824
The University of Manchester

Verification of concurrent programs

Testing and verifying concurrent programs is an inherently **difficult task**

- Different possible threads' interleavings make the program execution **non-deterministic**:
  - Some bugs may occur only for a specific **thread's order**

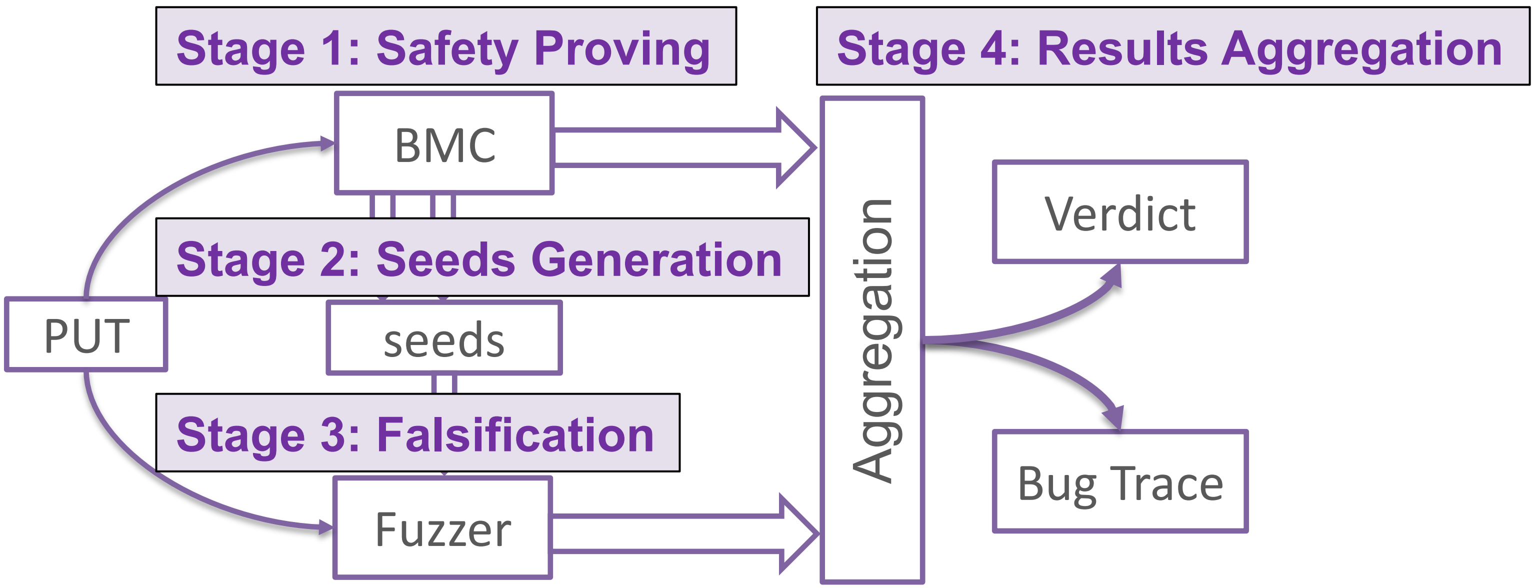
- Existing techniques often have various theoretical and practical **limitations**

The main idea of **cooperative verification** is to implement a **communication interface** between different tools, which allows the exchange of **partial verification results**

EBF Cooperative Approach

The University of Manchester

In **EBF** we implement an open-source[1] tool combining **BMC** and **fuzzing**



1. https://github.com/fatimahkj/EBF

MANCHESTER 1824
The University of Manchester

Stage 1: Safety Proving

- Here EBF calls the **BMC** engine for the given program.
  - It produces one of the three possible verdicts: **Safe, Bug**, or **Unknown**.
  - This is the **only** time when EBF can prove **program safety**

- If the BMC tool returns **Bug**, it generates a **counter-example**
  - a **sequence** of **program inputs** and a **thread schedule** leading to the vulnerability
  - all produced counter-examples are saved for further use

```
Counterexample:

State 5 file test.c line 9 function foo thread 1
----------------------------------------------------
  a = 42 (00000000 00000000 00000000 00101010)

State 6 file test.c line 10 function foo thread 1
----------------------------------------------------
  b = 0 (00000000 00000000 00000000 00000000)

State 7 file test.c line 5 function reach_error thread 1
----------------------------------------------------
Violated property:
  file test.c line 5 function reach_error
  assertion 0
  0


VERIFICATION FAILED
```

Stage 2: Seeds Generation

- This is introduced in **EBF 4.2**
- For each conditional branch (i.e., **if**, **else**, **while**, **for**, …) in the program:

1. Inject an error statement (i.e., **assert(0)**) inside the branch
2. Run the **BMC** tool on the newly **instrumented** program
3. If BMC returns **Bug**, then convert the **counter-example** into a **seed** for the fuzzer
4. Otherwise (**Safe**, **Unknown** or ***timeout***), move to the next branch in the program and go to Step 1.

- The seed generating continues until **all injected errors** have been detected or the **stage timeout** has been reached.

- The generated seeds greatly **improve** the fuzzer performance in the next stage.

Stage 3: Falsification

- **EBF** checks whether the program contains any vulnerabilities by **fuzzing**
- Out of the box fuzzers (i.e., **libFuzz**, **AFL**) are not suitable for testing concurrent programs
    - They do not have access to different **thread schedules**

- We implement and use **OpenGBF** – open-source grey-box fuzzer
    - Based on **AFL++** (thread-safe version of **AFL**)
    - It injects **delay functions** after every instruction in the program via an **LLVM pass**
    - Different **delay values** enforce different **thread schedules**
    - The **delay values** and the **program inputs** are "sampled" by **AFL++** using previously generated **seeds**
    - Other instrumentations are applied to generate counter-examples, ensure atomic execution, etc.

Stage 4: Results Aggregation

- **EBF** produces a verification **verdict** and a **bug trace** (if either tool returns **Bug**)

Results Aggregation Stage

		OpenGBF	
		Bug	Unknown
BMC	Safe	*Conflict*	*Safe*
	Bug	*Unsafe*	*Unsafe*
	Unknown	*Unsafe*	*Unknown*

- When one of the tools returns **Unknown**, **EBF** relies on the verdict of the other one
- When the **BMC** tool returns **Safe**, and **OpenGBF** outputs **Bug**, **EBF** reports **Conflict**
  - This requires analysing the bug trace produced by **OpenGBF**
  - The **BMC** tool can be wrong due to **over-approximations**
  - **OpenGBF** can be wrong due with respect to the given **property** (i.e., something else causes the crash)

EBF 4.0 with different BMC tools

The University of Manchester

Experimental Setup:

- **BMC** 6 min + **OpenGBF** 5 min + **results Aggregation** 4 min = 15 min.
- **RAM limit** is 15 GB per Benchexec run.
- **ConcurrencySafety main** from SV-COMP 2022.
  - Witness validation switched off.
- Ubuntu 20.04.4 LTS with 160 GB RAM and 25 cores

		EBF and BMC tools							
		EBF	Deagle	EBF	Cseq	EBF	ESBMC	EBF	CBMC
Results	Correct True	240	240	172	**177**	65	**70**	139	146
	Correct False	**336**	319	**333**	313	**308**	268	**320**	303
	Incorrect True	0	0	0	0	0	0	0	0
	Incorrect False	0	0	0	0	0	1	0	3
	Overall	**816**	799	**677**	667	**438**	376	**598**	547

- EBF4.0 **increases** the number of **found bugs** in comparison to the individual BMC tools.
- Overall, EBF4.0 provides a better **trade-off** between **bug finding** and **safety proving** than each BMC engine

The University of Manchester

EBF 4.2 in SV-COMP 2023

In EBF 4.2 we used **ESBMC** as BMC engine

- **ESBMC** 6 min + **Seed Generation** 1 min+ **OpenGBF** 5 min + **results Aggregation** 3 min = 15 min.

EBF 4.2 participated in concurrencySafety main

Results	EBF	ESBMC
Correct True	67	71
Correct False	251	236
Incorrect True	0	1
Incorrect False	1	0
Overall	369	346

MANCHESTER 1824
The University of Manchester

Limitations

1) **The order** of the values in the counter example is **not** always the same as their order in the program.
2) Some benchmarks can contain **multiple different bugs**, which is **fine** for static analysis tools (BMC) but **not suitable** for dynamic analysis tools (e.g., one bug is always triggered before the other).
3) EBF4.2 only offered partial support for **data race detection** because ESBMC does not yet maintain full support for this property.
4) EBF4.2 does not yet support the detection of **arithmetic overflows** and **memory safety violations** as required by the competition format.

The University of Manchester

Reference

[1] F. K. Aljaafari, R. Menezes, E. Manino, et al., "***Combining bmc and fuzzing techniques for finding software vulnerabilities in concurrent programs***," IEEE Access, vol. 10, pp. 121 365–121 384, 2022. doi: 10.1109/ACCESS.2022.3223359

[2] F. Aljaafari, F. Shmarov, E. Manino, et al., "***EBF 4.2: Black-Box cooperative verification for concurrent programs (competition contribution),***" in Proc. TACAS (2), ser. LNCS, Springer, 2023

Thank you